

Database Toolbox™

User's Guide



MATLAB®

R2023a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Database Toolbox™ User's Guide

© COPYRIGHT 1998–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

May 1998	Online Only	New for Version 1 for MATLAB® 5.2
July 1998	First Printing	For Version 1
Online only	June 1999	Revised for Version 2 (Release 11)
December 1999	Second printing	For Version 2 (Release 11)
Online only	September 2000	Revised for Version 2.1 (Release 12)
June 2001	Third printing	Revised for Version 2.2 (Release 12.1)
July 2002	Online only	Revised for Version 2.2.1 (Release 13)
November 2002	Fourth printing	Version 2.2.1
June 2004	Fifth printing	Revised for Version 3.0 (Release 14)
October 2004	Online only	Revised for Version 3.1 (Release 14SP1)
March 2005	Online only	Revised for Version 3.0.2 (Release 14SP2)
September 2005	Online only	Revised for Version 3.1 (Release 14SP3)
March 2006	Online only	Revised for Version 3.1.1 (Release 2006a)
September 2006	Online only	Revised for Version 3.2 (Release 2006b)
October 2006	Sixth printing	Revised for Version 3.2 (Release 2006b)
March 2007	Online only	Revised for Version 3.3 (Release 2007a)
September 2007	Seventh printing	Revised for Version 3.4 (Release 2007b)
March 2008	Online only	Revised for Version 3.4.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.5 (Release 2008b)
March 2009	Online only	Revised for Version 3.5.1 (Release 2009a)
September 2009	Online only	Revised for Version 3.6 (Release 2009b)
March 2010	Online only	Revised for Version 3.7 (Release 2010a)
September 2010	Online only	Revised for Version 3.8 (Release 2010b)
April 2011	Online only	Revised for Version 3.9 (Release 2011a)
September 2011	Online only	Revised for Version 3.10 (Release 2011b)
March 2012	Online only	Revised for Version 3.11 (Release 2012a)
September 2012	Online only	Revised for Version 4.0 (Release 2012b)
March 2013	Online only	Revised for Version 4.1 (Release 2013a)
September 2013	Online only	Revised for Version 5.0 (Release 2013b)
March 2014	Online only	Revised for Version 5.1 (Release 2014a)
October 2014	Online only	Revised for Version 5.2 (Release 2014b)
March 2015	Online only	Revised for Version 5.2.1 (Release 2015a)
September 2015	Online only	Revised for Version 6.0 (Release 2015b)
March 2016	Online only	Revised for Version 6.1 (Release 2016a)
September 2016	Online only	Revised for Version 7.0 (Release 2016b)
March 2017	Online only	Revised for Version 7.1 (Release 2017a)
September 2017	Online only	Revised for Version 8.0 (Release 2017b)
March 2018	Online only	Revised for Version 8.1 (Release 2018a)
September 2018	Online only	Revised for Version 9.0 (Release 2018b)
March 2019	Online only	Revised for Version 9.1 (Release 2019a)
September 2019	Online only	Revised for Version 9.2 (Release 2019b)
March 2020	Online only	Revised for Version 9.2.1 (Release 2020a)
September 2020	Online only	Revised for Version 10.0 (Release 2020b)
March 2021	Online only	Revised for Version 10.1 (Release 2021a)
September 2021	Online only	Revised for Version 10.2 (Release 2021b)
March 2022	Online only	Revised for Version 10.3 (Release 2022a)
September 2022	Online only	Revised for Version 10.4 (Release 2022b)
March 2023	Online only	Revised for Version 11.0 (Release 2023a)

1	Before You Begin
	<hr/>
	Database Toolbox Product Description 1-2
	Data Type Support 1-3
	Data Retrieval Restrictions 1-5
	Spaces in Table Names or Column Names 1-5
	Quotation Marks in Table Names or Column Names 1-5
	Reserved Words in Column Names 1-5

2	Getting Started with Database Toolbox
	<hr/>
	Access Relational Database Data in MATLAB 2-2
	Interact with Data in SQLite Database Using MATLAB Interface to SQLite 2-5
	MATLAB Interface to SQLite Advantages 2-5
	Command Line Workflow for MATLAB Interface to SQLite 2-5
	Database Explorer App Workflow for MATLAB Interface to SQLite 2-5
	SQLite JDBC Connection Differences 2-6
	MATLAB Interface to SQLite Limitations 2-7
	Connection Options 2-8
	Creating or Connecting to Data Source 2-8
	Defining Operating System Authentication 2-8
	Connection Options 2-8
	Setup Requirements for Database Connection 2-11
	Choose Between ODBC and JDBC Drivers 2-12
	Defining Database Drivers 2-12
	Deciding Between ODBC and JDBC Drivers 2-12
	Configure Driver and Data Source 2-14
	Configure Driver and Create Data Source 2-14
	Limitations 2-15
	Create JDBC Data Source and Set Options Programmatically 2-17

Microsoft Access ODBC for Windows	2-19
Step 1. Set up the sample Access database.	2-19
Step 2. Verify the driver installation.	2-19
Step 3. Set up the data source using the Database Explorer app.	2-19
Step 4. Connect using the Database Explorer app or the command line.	2-21
Microsoft SQL Server ODBC for Windows	2-23
Step 1. Verify the driver installation.	2-23
Step 2. Set up the data source using the Database Explorer app.	2-23
Step 3. Connect using the Database Explorer app or the command line.	2-26
Microsoft SQL Server ODBC for Windows DSN-Less Connection	2-28
Step 1. Verify the driver installation.	2-28
Step 2. Connect using the DSN-less connection string and command line.	2-28
Microsoft SQL Server JDBC for Windows	2-29
Step 1. Verify the driver installation.	2-29
Step 2. Verify the port number.	2-29
Step 3. Set up the operating system authentication.	2-31
Step 4. Set up the data source.	2-31
Step 5. Connect using the Database Explorer app or the command line.	2-34
Oracle ODBC for Windows	2-36
Step 1. Verify the driver installation.	2-36
Step 2. Set up the data source using the Database Explorer app.	2-36
Step 3. Connect using the Database Explorer app or the command line.	2-39
Oracle JDBC for Windows	2-41
Step 1. Verify the driver installation.	2-41
Step 2. Set up the operating system authentication.	2-41
Step 3. Set up the data source.	2-41
Step 4. Connect using the Database Explorer app or the command line.	2-45
MySQL ODBC for Windows	2-47
Step 1. Verify the driver installation.	2-47
Step 2. Set up the data source using the Database Explorer app.	2-47
Step 3. Connect using the Database Explorer app or the command line.	2-49
MySQL ODBC for Windows DSN-Less Connection	2-51
Step 1. Verify the driver installation.	2-51
Step 2. Connect using the DSN-less connection string and command line.	2-51
MySQL JDBC for Windows	2-52
Step 1. Verify the driver installation.	2-52
Step 2. Set up the data source.	2-52
Step 3. Connect using the Database Explorer app or the command line.	2-54

PostgreSQL ODBC for Windows	2-56
Step 1. Verify the driver installation.	2-56
Step 2. Set up the data source using the Database Explorer app.	2-56
Step 3. Connect using the Database Explorer app or the command line.	2-58
PostgreSQL ODBC for Windows DSN-Less Connection	2-60
Step 1. Verify the driver installation.	2-60
Step 2. Connect using the DSN-less connection string and command line.	2-60
PostgreSQL JDBC for Windows	2-61
Step 1. Verify the driver installation.	2-61
Step 2. Set up the data source.	2-61
Step 3. Connect using the Database Explorer app or the command line.	2-63
SQLite JDBC for Windows	2-65
Step 1. Verify the driver installation.	2-65
Step 2. Set up the data source.	2-65
Step 3. Connect using the Database Explorer app or the command line.	2-68
Microsoft SQL Server JDBC for macOS	2-70
Step 1. Verify the driver installation.	2-70
Step 2. Set up the data source.	2-70
Step 3. Connect using the Database Explorer app or the command line.	2-72
Microsoft SQL Server JDBC for Linux	2-74
Step 1. Verify the driver installation.	2-74
Step 2. Set up the data source.	2-74
Step 3. Connect using the Database Explorer app or the command line.	2-76
Microsoft SQL Server ODBC for Linux	2-78
Step 1. Verify the driver installation.	2-78
Step 2. Set up the data source.	2-78
Step 3. Connect using the command line.	2-78
Microsoft SQL Server ODBC for Linux DSN-Less Connection	2-79
Step 1. Verify the driver installation.	2-79
Step 2. Connect using the DSN-less connection string and command line.	2-79
Oracle JDBC for macOS	2-80
Step 1. Verify the driver installation.	2-80
Step 2. Set up the data source.	2-80
Step 3. Connect using the Database Explorer app or the command line.	2-83
Oracle JDBC for Linux	2-85
Step 1. Verify the driver installation.	2-85
Step 2. Set up the data source.	2-85

Step 3. Connect using the Database Explorer app or the command line.	2-88
MySQL JDBC for macOS	2-90
Step 1. Verify the driver installation.	2-90
Step 2. Set up the data source.	2-90
Step 3. Connect using the Database Explorer app or the command line.	2-92
MySQL ODBC for Linux	2-94
Step 1. Verify the driver installation.	2-94
Step 2. Set up the data source.	2-94
Step 3. Connect using the command line.	2-94
MySQL ODBC for Linux DSN-Less Connection	2-95
Step 1. Verify the driver installation.	2-95
Step 2. Connect using the DSN-less connection string and command line.	2-95
MySQL JDBC for Linux	2-96
Step 1. Verify the driver installation.	2-96
Step 2. Set up the data source.	2-96
Step 3. Connect using the Database Explorer app or the command line.	2-98
PostgreSQL JDBC for macOS	2-100
Step 1. Verify the driver installation.	2-100
Step 2. Set up the data source.	2-100
Step 3. Connect using the Database Explorer app or the command line.	2-102
PostgreSQL ODBC for Linux	2-104
Step 1. Verify the driver installation.	2-104
Step 2. Set up the data source.	2-104
Step 3. Connect using the command line.	2-104
PostgreSQL ODBC for Linux DSN-Less Connection	2-105
Step 1. Verify the driver installation.	2-105
Step 2. Connect using the DSN-less connection string and command line.	2-105
PostgreSQL JDBC for Linux	2-106
Step 1. Verify the driver installation.	2-106
Step 2. Set up the data source.	2-106
Step 3. Connect using the Database Explorer app or the command line.	2-108
SQLite JDBC for macOS	2-110
Step 1. Verify the driver installation.	2-110
Step 2. Set up the data source.	2-110
Step 3. Connect using the Database Explorer app or the command line.	2-112
SQLite JDBC for Linux	2-114
Step 1. Verify the driver installation.	2-114

Step 2. Set up the data source.	2-114
Step 3. Connect using the Database Explorer app or the command line.	2-116
Other ODBC-Compliant or JDBC-Compliant Databases	2-118
ODBC-Compliant Databases	2-118
JDBC-Compliant Databases	2-118
Connect to Database	2-121
Database Explorer App Connection Workflow	2-121
Command Line Connection Workflow	2-121
Database List	2-121
Data Import Using Database Explorer App or Command Line	2-125
Data Import Using Database Explorer App	2-125
Data Import Using Command Line	2-125
Custom Data Types	2-126
SQL Queries Saved in Scripts or Files	2-126
Working with Large Data Sets	2-127
Connect to a Database with Maximum Performance	2-127
Import Large Data Sets into MATLAB	2-127
Export Large Data Sets from MATLAB	2-127
Access Large Data Using a DatabaseDatastore	2-127

Working with Data Sources

3

Writing Data Common Errors	3-2
Importing Data Common Errors	3-3
Data Import Common Errors	3-3
Custom Import Options Common Errors	3-5
Database Connection Error Messages	3-7
Database Explorer App Error Messages	3-13
SQL Prepared Statement Error Messages	3-15

Using Database Explorer

4

Create SQL Queries Using Database Explorer App	4-2
Create SQL Query Using Toolstrip Buttons	4-2
Enter SQL Query Manually	4-4
Work with Multiple SQL Queries	4-5
SQL Query Limitations	4-5

Customize Import Options Using Database Explorer App	4-7
Join Tables Using Database Explorer App	4-10
Different Join Types	4-10
Join Tables	4-10
Join Diagram	4-12
Join Type Limitations	4-13
Data Preview Using Database Explorer App	4-14
Automatic Preview	4-14
Preview Size	4-14
Preview Data by Creating SQL Query	4-15
Preview Data by Entering SQL Query Manually	4-16
Modify and Delete Data Sources	4-17
Modify Data Sources Interactively	4-17
Modify Data Sources Programmatically	4-17
Delete Data Sources Interactively	4-18
Delete Data Sources Programmatically	4-18
Generate SQL Query and MATLAB Script	4-20
Generate SQL Query	4-20
Generate MATLAB Script	4-20

Using Database Toolbox Functions

5

Roll Back Data in Database	5-2
Change Database Connection Catalog	5-4
Create Table and Add Column	5-5
Delete Data from Databases	5-6
Roll Back Data After Updating Record	5-9
Replace Existing Data in Database	5-12
Export Data Using Bulk Insert	5-13
Bulk Insert Functionality	5-13
Bulk Insert into Oracle	5-13
Bulk Insert into Microsoft SQL Server 2005	5-14
Bulk Insert into MySQL	5-15
Call Stored Procedure That Returns Data	5-17
Run Custom Database Function	5-19
Data Import Memory Management	5-20
sqlread Function	5-20
select Function	5-20

Define Import Strategy Using SQLImportOptions Object	5-21
Import Large Data Using DatabaseDatastore Object	5-23
Analyze Large Data in Database Using MapReduce	5-27
Analyze Large Data in Database Using Tall Arrays	5-30
Import Data Using MATLAB Interface to SQLite	5-32
Insert Data into SQLite Database Table	5-36
Create Table and Add Column in SQLite Database	5-38
Delete Data from SQLite Database	5-39
Roll Back Data in SQLite Database	5-41
Deploy MATLAB Interface to SQLite Database Application with MATLAB Compiler	5-43
Retrieve Image Data Types	5-48
Import Boolean Data from Database	5-51
Append Data to Existing Database Table Using Insert Functionality ...	5-53
Insert Data into New Database Table Using Insert Functionality	5-55
Join Tables Using Command Line	5-57
Import Data from Database Table Using sqlread Function	5-58
Insert Data into Database Table	5-61
Retrieve Database Metadata	5-64
Customize Options for Importing Data from Database into MATLAB ...	5-67
Deploy Relational Database Application with MATLAB Compiler	5-70
Import Data Using SQL Prepared Statement with Multiple Parameter Values	5-75

MySQL Native Interface Topics

6

Configure MySQL Native Interface Data Source	6-2
Step 1. Set up the data source.	6-2
Step 4. Connect using the Database Explorer app or the command line.	6-4

Import Data from MySQL Database Table	6-6
Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface	6-8
Import Large Data Using DatabaseDatastore Object and MySQL Native Interface	6-11
Insert Data into Database Table Using MySQL Native Interface	6-14
Roll Back Data in Database Using MySQL Native Interface	6-16
Create Table and Add Column Using MySQL Native Interface	6-18
Delete Data from Database Using MySQL Native Interface	6-19
Deploy MySQL Native Interface Database Application with MATLAB Compiler	6-21

PostgreSQL Native Interface Topics

7

Configure PostgreSQL Native Interface Data Source	7-2
Step 1. Set up the data source.	7-2
Step 2. Connect using the Database Explorer app or the command line.	7-4
Deploy PostgreSQL Native Interface Database Application with MATLAB Compiler	7-6
Import Data from PostgreSQL Database Table	7-11
Customize Options for Importing Data from PostgreSQL Database into MATLAB	7-13
Import Large PostgreSQL Data Using DatabaseDatastore Object	7-16
Insert Data into Database Table Using PostgreSQL Native Interface ...	7-19
Roll Back Data in Database Using PostgreSQL Native Interface	7-21
Create Table and Add Column Using PostgreSQL Native Interface	7-23
Delete Data from Database Using PostgreSQL Native Interface	7-24

Apache Cassandra Database C++ Interface Topics

8

Convert CQL Data Types to MATLAB Data Types Using Apache Cassandra Database C++ Interface	8-2
Fill Values for Missing Data from Database Using Apache Cassandra Database C++ Interface	8-9
Explore and Import Data from Cassandra Database Table	8-11
Import Data from Cassandra Database Table Using CQL	8-14
Export MATLAB Data into Cassandra Database	8-16

Neo4j Topics

9

Explore Graph Database Structure	9-2
Graph Database Workflow for Neo4j Database Interfaces	9-6
About Neo4j Graph Databases	9-6
Neo4j Graph Database Workflow	9-6
Advantage of Database Toolbox Interface for Neo4j Bolt Protocol	9-7
Search Graph Database	9-9
Search Functionality	9-9
General and Targeted Search Workflows	9-9
Update Friend Information in Social Neighborhood	9-11
Add and Query Group of Colleagues in Social Neighborhood	9-14
Error Messages for Neo4j Database Interfaces	9-20
Determine Dependencies of Services in Network	9-22
Find Shortest Path Between People in Social Neighborhood	9-27
Find Friends of Friends in Social Neighborhood	9-32
Database Toolbox Interface for Neo4j Bolt Protocol Installation	9-37
Supported Neo4j Versions	9-37
Installation	9-37
Deploy Graph Database Application with MATLAB Compiler	9-38

10

Import and Analyze Data from MongoDB Using MongoDB C++ Interface	10-2
Import Filtered Data from MongoDB Using MongoDB C++ Interface	10-4
Import Large Data from MongoDB Using MongoDB C++ Interface	10-6
Export MATLAB Data into MongoDB Using MongoDB C++ Interface	10-8
Import and Export MATLAB Objects Using MongoDB and MongoDB C++ Interface	10-10

Functions

11

Before You Begin

Database Toolbox Product Description

Interact with relational and NoSQL databases

Database Toolbox provides functions and an app for interacting with relational and NoSQL databases. The toolbox supports any ODBC-compliant or JDBC-compliant relational database and provides NoSQL support for Apache® Cassandra®, MongoDB®, and Neo4j®.

Native interfaces for MySQL®, PostgreSQL, and SQLite provide functionality that streamlines interaction with your database. The interfaces ship with third-party drivers that connect to your database in deployed and cloud environments without additional setup or installation.

With the Database Explorer app, you can explore relational data interactively and generate MATLAB® code to automate or operationalize database workflows. For large data workflows, you can run query filters directly on your SQL queries.

Data Type Support

You can import these data types into the MATLAB workspace and export them back to your database.

Database	Supported Data Type
All databases	<ul style="list-style-type: none"> • BOOLEAN • CHAR • DATE • DECIMAL • DOUBLE • FLOAT • INTEGER • NUMERIC • REAL • SMALLINT • TIME • TIMESTAMP <p>Note When importing TIMESTAMP data into MATLAB, you can get an incorrect value at the daylight saving time change. To avoid an incorrect value, convert TIMESTAMP data to strings in your SQL query. Then, convert the strings back to the MATLAB data type you want. Or, connect to your database using a different driver.</p>
Microsoft® SQL Server®	<ul style="list-style-type: none"> • NTEXT • TEXT • VARCHAR (MAX) • CHAR (8000) • NCHAR (4000) • NVARCHAR (MAX) • TINYINT
MySQL	<ul style="list-style-type: none"> • MEDIUMTEXT • LONGTEXT • TINYINT
Oracle®	<ul style="list-style-type: none"> • LONG • VARCHAR2 (4000) • NCHAR (2000) • NVARCHAR2 (4000)

To import data of another data type, manipulate the data before importing it into the MATLAB workspace. For details, see your database documentation.

See Also

`fetch` | `sqlwrite` | `update`

More About

- “Import Data from Database Table Using `sqlread` Function” on page 5-58
- “Insert Data into Database Table” on page 5-61

Data Retrieval Restrictions

Spaces in Table Names or Column Names

Microsoft Access® supports the use of spaces in table and column names, but most other databases do not. Queries that retrieve data from tables and fields whose names contain spaces require delimiters around table names and field names. In Access®, enclose the table names or field names in quotation marks, for example, "order id". Other databases use different delimiters, such as brackets, [].

Quotation Marks in Table Names or Column Names

Do not include quotation marks in table names or column names. The Database Toolbox software does not support data retrieval from table and column names that contain quotation marks.

Reserved Words in Column Names

You cannot use the Database Toolbox software to import or export data in columns whose names contain database reserved words, such as DATE or TABLE.

See Also

More About

- "Data Import Using Database Explorer App or Command Line" on page 2-125

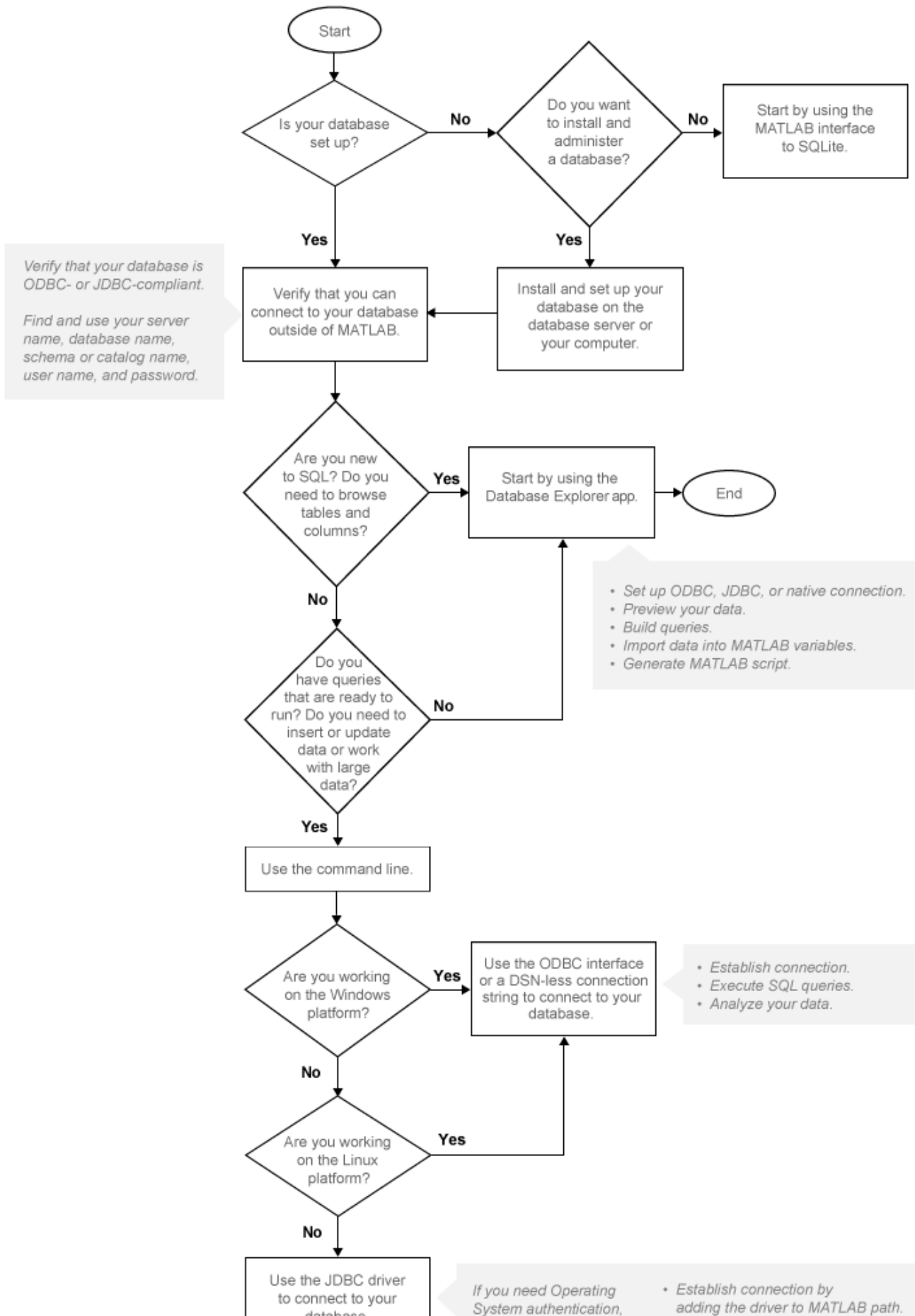
Getting Started with Database Toolbox

Access Relational Database Data in MATLAB

This tutorial shows how to use Database Toolbox with relational databases. To gain the maximum benefit from this toolbox and understand its capabilities, use the following steps and decision flow chart.

- 1** If you do not have an installed database and want to store relational data quickly, use the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.
- 2** Install your database. For details, refer to your database administrator or your database documentation.
- 3** Choose between using the **Database Explorer** app or the command line. For details, see “Connection Options” on page 2-8.
- 4** If you have a MySQL database, then you can use the MySQL native interface. For details, see “MySQL Native Interface”.
- 5** If you have a PostgreSQL database, then you can use the PostgreSQL native interface. For details, see “PostgreSQL Native Interface”.
- 6** Choose between using an ODBC driver or a JDBC driver. For details, see “Choose Between ODBC and JDBC Drivers” on page 2-12.
- 7** If you are using a JDBC driver, you need to install the driver. An ODBC driver is typically preinstalled on your computer. For details about ODBC and JDBC drivers, see Driver Installation. If you have questions about which driver you need, refer to your database administrator or your database documentation.
- 8** If you have a Linux® platform, then you can use the `odbc` function to create an ODBC database connection using a defined data source or a connection string. The connection string is a DSN-less connection. (DSN is a data source name.)
- 9** Create a data source for an ODBC-compliant or JDBC-compliant driver. For details, see “Configure Driver and Data Source” on page 2-14.
- 10** Test the connection to your database using the Database Explorer app or the command line.
- 11** Connect to your database using the Database Explorer app or the command line. For details, see “Connect to Database” on page 2-121.
- 12** Select data from your database and import the data into a MATLAB variable by using the Database Explorer app or the command line. For details, see “Data Import Using Database Explorer App or Command Line” on page 2-125.
- 13** Insert data into your database by exporting data from a MATLAB variable. For details, see the `sqlwrite` function.
- 14** When you use the Database Explorer app to import data, generate a MATLAB script to automate your tasks. For details, see “Generate MATLAB Script” on page 4-20.

This flow chart illustrates the steps to take and the decisions to make when you use Database Toolbox with relational databases.



See Also

More About

- “Setup Requirements for Database Connection” on page 2-11
- “Choose Between ODBC and JDBC Drivers” on page 2-12
- “Configure Driver and Data Source” on page 2-14
- “Connect to Database” on page 2-121
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5

Interact with Data in SQLite Database Using MATLAB Interface to SQLite

To analyze your data using SQL in MATLAB without access to any database or driver, use the MATLAB interface to SQLite. After installing Database Toolbox, you can use the MATLAB interface to SQLite to move data between MATLAB and an SQLite database file. For details, see “MATLAB Interface to SQLite”. The SQLite connection is different from a database connection created using a JDBC driver. For background information about SQLite databases, see the SQLite Home Page.

MATLAB Interface to SQLite Advantages

With the MATLAB interface to SQLite, you can start working with data immediately after installing the Database Toolbox by creating an SQLite database file. No installation or administration of software or drivers is required. You can share data using SQLite database files. Also, you can deploy a standalone database application without additional installation or configuration on target machines. The MATLAB interface to SQLite supports Windows®, Linux, and Mac.

Command Line Workflow for MATLAB Interface to SQLite

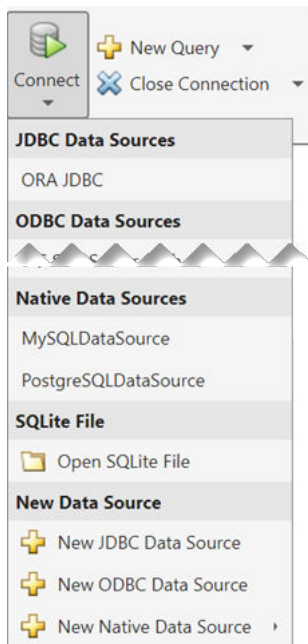
To connect to a database quickly and import data, use the MATLAB interface to SQLite. These steps provide a high-level workflow for using the MATLAB interface to SQLite.

- 1 Create an SQLite database file by using `sqlite`. The SQLite database file has a `.db` extension. The `sqlite` object signifies an SQLite database connection.
- 2 Create tables in the SQLite database file by using `execute`.
- 3 Export your data into the SQLite database file by using `sqlwrite`.
- 4 Import data into MATLAB by using `sqlread` or `fetch`.
- 5 Perform data analysis in MATLAB.
- 6 Export results into the SQLite database file by using `sqlwrite`.
- 7 Close the SQLite connection by using `close`.
- 8 Share the SQLite database file with others.
- 9 Create a standalone SQLite database application and deploy it.

Database Explorer App Workflow for MATLAB Interface to SQLite

If you have an existing SQLite database file, you can use the **Database Explorer** app to create an SQLite connection. These steps show how to create the connection.

- 1 Open the Database Explorer app.
- 2 In the **Connections** section, click **Connect**.
- 3 Select **Open SQLite File**.



- 4 Navigate to the folder where your SQLite database file exists and select the file. Click **OK**.
- 5 Database Explorer creates an SQLite connection. The **Database Browser** pane displays the available tables in the database.

After you create the SQLite connection, you can use Database Explorer to explore the data in the SQLite database and import the data into MATLAB. For details, see these topics:

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14

SQLite JDBC Connection Differences

The following table describes the differences between using the MATLAB interface to SQLite and using the JDBC driver to connect to an SQLite database.

Item	SQLite Connection Using the MATLAB Interface to SQLite	SQLite Database Connection Using a JDBC Driver
Driver installation	Not required	Required
Database installation	Not required	Required
Database administration	Not required	Required
Database connection function	sqlite	database
Import data	Yes	Yes
Export data	Yes	Yes
Database operations	Yes	Yes
Database Explorer	Yes	Yes

Item	SQLite Connection Using the MATLAB Interface to SQLite	SQLite Database Connection Using a JDBC Driver
Database metadata	No	Yes
Complex functionality	No	Yes

MATLAB Interface to SQLite Limitations

The limitations of using the MATLAB interface to SQLite are:

- NULL values are not supported as the first value in any column.
- Database import options are not supported.

See Also

Related Examples

- “Access Relational Database Data in MATLAB” on page 2-2
- “Import Data Using MATLAB Interface to SQLite” on page 5-32
- “Insert Data into SQLite Database Table” on page 5-36
- “Create Table and Add Column in SQLite Database” on page 5-38
- “Deploy MATLAB Interface to SQLite Database Application with MATLAB Compiler” on page 5-43

External Websites

- [SQLite Home Page](#)

Connection Options

You can connect to a database in various ways using Database Toolbox. If you have access to a database, create a data source. Then, you can connect to the database either by using the **Database Explorer** app or the command line. If you do not have an installed database and want to store relational data quickly, you can use the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Creating or Connecting to Data Source

If you already have a database driver installed, you can create a data source.

- For an ODBC driver, open the Microsoft ODBC Data Source Administrator dialog box by using the `configureODBCDataSource` function or the Database Explorer app.
- For a JDBC driver, open the JDBC Data Source Configuration dialog box by using the Database Explorer app, or configure a JDBC data source by using the `databaseConnectionOptions` function at the command line.

For examples, see “Configure Driver and Data Source” on page 2-14. Otherwise, see Driver Installation for information about installing your driver. For configuring MySQL or PostgreSQL native interfaces, see “MySQL Native Interface” or “PostgreSQL Native Interface”.

If your data source is already defined, then you are ready to connect to your database. If you do not define a data source, you can connect to your database using a DSN-less connection string with the `odbc` function. (DSN is a data source name.) For details, see “Connect to Database” on page 2-121.

After establishing a connection, you can explore the database and view data using the Database Explorer app or the command line. For details, see “Data Import Using Database Explorer App or Command Line” on page 2-125.

Defining Operating System Authentication

Operating system authentication enables you to connect to a database using your operating system user account. The operating system performs user validation, and the database does not require a different user name and password. Operating system authentication facilitates the maintenance of database access credentials. For example, Windows provides operating system authentication that can be configured to work with a Microsoft SQL Server database. For details about Microsoft SQL Server Windows authentication, see Set up the operating system authentication. on page 2-31

Connection Options

Use this table to choose your best connection option.

Connection Option	Usage
Database Explorer app	<ul style="list-style-type: none"> • Visually inspect the structure, or schema, of a database. • Assess the general size of a database by viewing the database structure. • Select the data in a table and import it into a MATLAB variable. • Generate a MATLAB script. • Generate an SQL query.
Command line	<ul style="list-style-type: none"> • Import data from a database into MATLAB. • Export data from MATLAB into a database. • Work with large amounts of data. • Run SQL queries stored in text files. • Run stored procedures and functions.

Several options enable you to connect to your database at the command line. Use this table to choose your best command line connection option.

Command Line Connection Option	Usage
ODBC connection	Connect to your database with maximum performance.
DSN-less connection	Connect to your database by using a connection string. For details, see the <code>odbc</code> function.
Native interface connection	Connect to MySQL and PostgreSQL databases. For PostgreSQL databases, you can connect without the installation of a driver. For details, see “MySQL Native Interface” and “PostgreSQL Native Interface”.
JDBC connection	Achieve maximum platform independence.
SQLite connection	Import data without installing a database or driver. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

You can create multiple connections at a time to the same database or different databases. For creating multiple connections using the Database Explorer app, see “Create SQL Queries Using Database Explorer App” on page 4-2. Or, you can use the `database` function to create multiple connections at the command line.

See Also

`odbc` | `database` | `mysql` | `postgresql`

More About

- “Choose Between ODBC and JDBC Drivers” on page 2-12
- “Connect to Database” on page 2-121
- “Data Import Using Database Explorer App or Command Line” on page 2-125
- “MySQL Native Interface”

- “PostgreSQL Native Interface”
- “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5

Setup Requirements for Database Connection

Refer to these setup requirements to establish the first connection to a database.

- If you do not have an installed database and want to store relational data quickly, use the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.
- Ensure that you know the name of your database server or machine, the name of your database, the port number, and your user name and password. For ODBC drivers, once you create a data source, remember the data source name. For JDBC drivers, ensure that you know the file path where the JDBC driver is installed. For some JDBC drivers, you also need the URL string and the driver Java® class object. For some databases, more credentials are required. Contact your database administrator for all required database credentials to establish a connection to the database.
- Ensure that you have access to your database and driver documentation.
- Determine if your database uses operating system authentication. If you can connect to your database from outside of MATLAB without providing a user name and password, then your database uses operating system authentication. Exceptions to this rule are databases set up without any operating system or database authentication requirements, such as Microsoft Access and SQLite database files. Connecting to a database using operating system authentication from MATLAB can require additional steps.
- Ensure that you have write access to the path MATLAB displays after you execute `prefdir` at the command line.
- For ODBC data sources, ensure that you have administrator rights to the database for creating and modifying data sources.

See Also

database

More About

- “Access Relational Database Data in MATLAB” on page 2-2
- “Connect to Database” on page 2-121

Choose Between ODBC and JDBC Drivers

Defining Database Drivers

Database vendors, such as Microsoft and Oracle, implement their database systems using technologies that vary depending on customer needs, market demands, and other factors. Software applications written in popular programming languages, such as C, C++, and Java, need a way to communicate with these databases. Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC) are standards for drivers that enable programmers to write database-agnostic software applications. ODBC and JDBC provide a set of rules recommended for efficient communication with a database. The database vendor is responsible for implementing and providing drivers that follow these rules.

Deciding Between ODBC and JDBC Drivers

ODBC is a standard Microsoft Windows interface that enables communication between database management systems and applications typically written in C or C++.

JDBC is a standard interface that enables communication between database management systems and applications written in Oracle Java.

Database Toolbox has a C++ library that connects natively to an ODBC driver. Database Toolbox has a Java library that connects directly to a pure JDBC driver.

Depending on your environment and what you want to accomplish, decide whether using an ODBC driver or a JDBC driver meets your needs.

Use native ODBC for:

- Fastest performance for data imports and exports
- Memory-intensive data imports and exports
- All functionality except the `runstoredprocedure` function

Use JDBC for:

- Platform independence, allowing you to work with any operating system (including Mac and Linux), driver version, or bitness
- Access to all Database Toolbox functions

The only limitation for these drivers is memory performance. MATLAB memory restricts the native ODBC driver. However, both MATLAB and JVM™ heap memory restrict the JDBC driver.

See Also

`database` | `close`

More About

- “Access Relational Database Data in MATLAB” on page 2-2
- “Connection Options” on page 2-8
- “Configure Driver and Data Source” on page 2-14

- “Connect to Database” on page 2-121
- “Working with Large Data Sets” on page 2-127

Configure Driver and Data Source

Database Toolbox enables you to create a connection to a database installed on a machine or located in the cloud. You can connect to relational databases using ODBC and JDBC drivers, or you can connect to some databases using native interfaces. This topic provides information about configuring a driver and creating a data source for connecting to a relational database using a driver. For details about connecting to a database using a native interface, see “Connection Options” on page 2-8.

To connect to an installed database, first install the driver. For all databases, define a data source for an ODBC or JDBC driver. If you do not have an installed database and want to store relational data quickly, you can use the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

ODBC uses a Data Source Name (DSN) that is the logical name to refer to the driver and other required information for accessing data. This name is used to connect to an ODBC data source, such as a Microsoft SQL Server database.

Configure Driver and Create Data Source

Find your database environment in the following table by choosing your platform across the top and your database on the left. The link brings you to a page that has all the required steps for connecting to your database.

Database	Platform		
	Windows	macOS 64-bit	Linux 64-bit
Microsoft Access	“Microsoft Access ODBC for Windows” on page 2-19	Not supported	Not supported
Microsoft SQL Server	“Microsoft SQL Server ODBC for Windows” on page 2-23 “Microsoft SQL Server ODBC for Windows DSN-Less Connection” on page 2-28 “Microsoft SQL Server JDBC for Windows” on page 2-29	“Microsoft SQL Server JDBC for macOS” on page 2-70	“Microsoft SQL Server ODBC for Linux” on page 2-78 “Microsoft SQL Server ODBC for Linux DSN-Less Connection” on page 2-79 “Microsoft SQL Server JDBC for Linux” on page 2-74
Oracle	“Oracle ODBC for Windows” on page 2-36 “Oracle JDBC for Windows” on page 2-41	“Oracle JDBC for macOS” on page 2-80	“Oracle JDBC for Linux” on page 2-85

Database	Platform		
	Windows	macOS 64-bit	Linux 64-bit
MySQL	<p>“MySQL ODBC for Windows” on page 2-47</p> <p>“MySQL ODBC for Windows DSN-Less Connection” on page 2-51</p> <p>“MySQL JDBC for Windows” on page 2-52</p>	“MySQL JDBC for macOS” on page 2-90	<p>“MySQL ODBC for Linux” on page 2-94</p> <p>“MySQL ODBC for Linux DSN-Less Connection” on page 2-95</p> <p>“MySQL JDBC for Linux” on page 2-96</p>
PostgreSQL	<p>“PostgreSQL ODBC for Windows” on page 2-56</p> <p>“PostgreSQL ODBC for Windows DSN-Less Connection” on page 2-60</p> <p>“PostgreSQL JDBC for Windows” on page 2-61</p>	“PostgreSQL JDBC for macOS” on page 2-100	<p>“PostgreSQL ODBC for Linux” on page 2-104</p> <p>“PostgreSQL ODBC for Linux DSN-Less Connection” on page 2-105</p> <p>“PostgreSQL JDBC for Linux” on page 2-106</p>
SQLite	“SQLite JDBC for Windows” on page 2-65	“SQLite JDBC for macOS” on page 2-110	“SQLite JDBC for Linux” on page 2-114

For ODBC-compliant or JDBC-compliant databases that are not listed in the table, see “Other ODBC-Compliant or JDBC-Compliant Databases” on page 2-118.

Limitations

The ODBC database connection on the Linux platform does not support the following:

- select function
- ODBC database connection using the **Database Explorer** app
- MySQL ODBC driver 8.0 and higher

See Also

database | close

More About

- “Access Relational Database Data in MATLAB” on page 2-2
- “Setup Requirements for Database Connection” on page 2-11
- “Choose Between ODBC and JDBC Drivers” on page 2-12

- “Connect to Database” on page 2-121
- “Modify and Delete Data Sources” on page 4-17
- “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5

Create JDBC Data Source and Set Options Programmatically

This example shows how to create a JDBC data source at the command line, set the JDBC connection options, and save the data source. The example configures a data source for a Microsoft® SQL Server® database.

Create JDBC Data Source

Create an SQL Server data source for a JDBC database connection.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc", vendor)

opts =
  SqlConnectionOptions with properties:
      DataSourceName: ""
      Vendor: "Microsoft SQL Server"
      JDBCDriverLocation: ""
      DatabaseName: ""
      Server: "localhost"
      PortNumber: 1433
      AuthenticationType: "Server"
```

opts is an SqlConnectionOptions object with these properties:

- DataSourceName — Name of the data source
- Vendor — Database vendor name
- JDBCDriverLocation — Full path of the JDBC driver file
- DatabaseName — Name of the database
- Server — Name of the database server
- PortNumber — Port number
- AuthenticationType — Authentication type

Set JDBC Connection Options

Configure the data source by setting the JDBC connection options for the data source SQLServerDataSource, full path to the JDBC driver file, database name toystore_doc, database server dbtb04, port number 54317, and Windows® authentication. Also, set driver-specific connection options to specify a timeout value for establishing the database connection, and to disable SSL encryption.

```
opts = setoptions(opts, ...
  "DataSourceName", "SQLServerDataSource", ...
  "JDBCDriverLocation", ...
  "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar", ...
  "DatabaseName", "toystore_doc", ...
  "Server", "dbtb04", "PortNumber", 54317, ...
  "AuthenticationType", "Windows", "loginTimeout", "20", ...
  "encrypt", "false")
```

```
opts =
    SqlConnectionOptions with properties:

        DataSourceName: "SQLServerDataSource"
        Vendor: "Microsoft SQL Server"

        JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
        DatabaseName: "toystore_doc"
        Server: "dbtb04"
        PortNumber: 54317
        AuthenticationType: "Windows"

    Additional Connection Options:

        encrypt: "false"
        loginTimeout: "20"
```

The `setoptions` function sets the `DataSourceName`, `JDBCDriverLocation`, `DatabaseName`, `Server`, `PortNumber`, and `AuthenticationType` properties in the `SqlConnectionOptions` object. The driver-specific connection options appear below the other connection options.

Test and Save JDBC Data Source

Test the database connection with a blank user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts,username,password)

status = logical
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `database` function or the Database Explorer app.

See Also

Objects

`SqlConnectionOptions`

Functions

`databaseConnectionOptions` | `setoptions` | `saveAsDataSource` | `database` | `testConnection`

More About

- “Modify and Delete Data Sources” on page 4-17
- “Configure Driver and Data Source” on page 2-14

Microsoft Access ODBC for Windows

This tutorial shows how to set up a data source and connect to a Microsoft Access database using the Database Explorer app or the command line. This tutorial uses the Microsoft Access Driver (*.mdb, *.accdb) to connect to a sample Microsoft Access 2016 database.

Step 1. Set up the sample Access database.

You can access the sample database file, `tutorial.accdb`, in the folder returned by entering this code at the command line.

```
fullfile(matlabroot, 'toolbox', 'database', 'dbdata')
```

Copy this database file into a folder where you have permission to write. Ensure that the database file is writable by verifying its properties:

- 1 Right-click the database file and select **Properties**.
- 2 On the **General** tab, if the **Read-only** option is selected, clear it.

Note To write data to the sample database, ensure that you run MATLAB as an administrator.

Note Depending on the Access version you are running, you might need to convert the database to that version. For example, beginning in Access 2007, the software includes the option to save as *.accdb. For details, consult your database administrator.

Step 2. Verify the driver installation.

The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers. For information about the Microsoft ODBC Data Source Administrator, see Driver Installation.

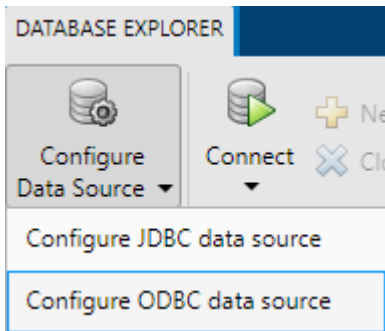
Note Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of Access. Or, to connect to the 32-bit version of Access, see <https://www.mathworks.com/matlabcentral/answers/235949-how-to-connect-to-32-bit-microsoft-access-database-from-64-bit-matlab>. For details about working with the 64-bit version of Windows, see Using Previous MATLAB Releases.

Step 3. Set up the data source using the Database Explorer app.

Set up the sample Access database as the data source by using the Database Explorer app. You can locate the target database on a PC running the Windows operating system or on another system to which the PC is networked. These instructions use the Microsoft ODBC Data Source Administrator Version 10.0.16299.15 for the US English version of Microsoft Access 2016 for Windows systems.

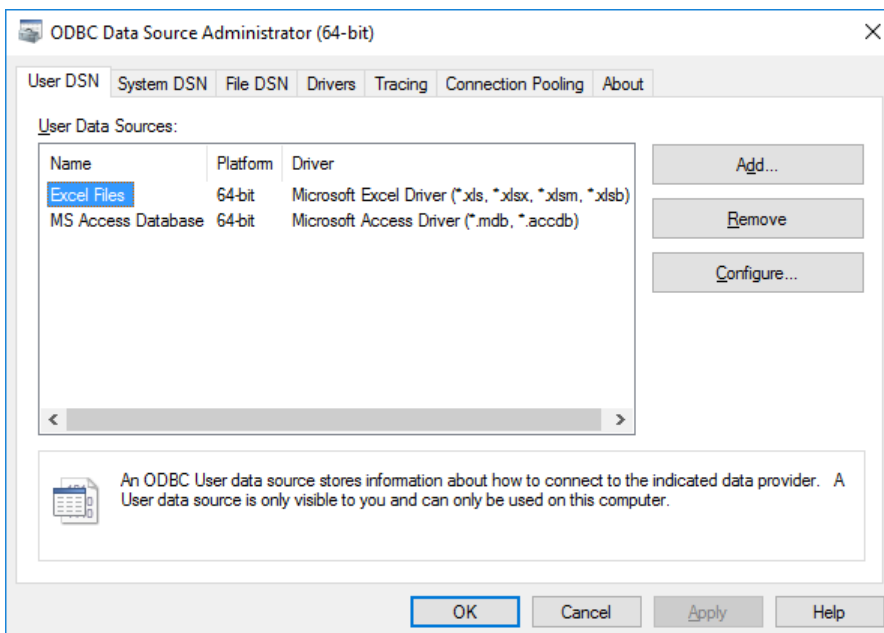
The Database Explorer app accesses the Microsoft ODBC Data Source Administrator automatically when you configure an ODBC data source. Alternatively, you can access the Microsoft ODBC Data Source Administrator using the `configureODBCDataSource` function.

- 1 Close any open Access databases.
- 2 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 3 In the **Data Source** section, select **Configure Data Source > Configure ODBC data source**.



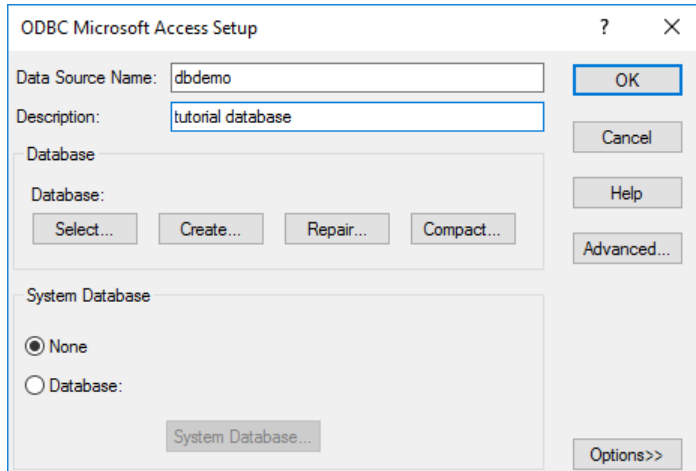
In the ODBC Data Source Administrator dialog box, you define the ODBC data source.

Tip When setting up an ODBC data source, you can specify a user data source name (DSN) or a system DSN. A user DSN is specific to the person logged into a machine. Only this person sees the data sources that are defined on the user DSN tab. A system DSN is not specific to the person logged into a machine. Any person who logs into the machine can see the data sources that are defined on the system DSN tab. Your ability to set up a user DSN or system DSN depends on the database and ODBC driver you are using. For details, contact the database administrator or refer to the ODBC driver documentation.



- 4 On the **User DSN** tab, click **Add**. The Create New Data Source dialog box opens and displays a list of installed ODBC drivers.

- 5 Select **Microsoft Access Driver (*.mdb, *.accdb)** and click **Finish**.
- 6 In the ODBC Microsoft Access Setup dialog box for your driver, enter `dbdemo` as the data source name. Enter `tutorial database` as the description.



- 7 Click **Select** to open the Select Database dialog box, where you specify the database you want to use. For the `dbdemo` data source, select `tutorial.accdb`. If the database is on a system to which your PC is connected:
 - a Click **Network**.
 - b In the Map Network Drive dialog box, specify the folder containing the database you want to use. Ensure that you map to the *folder* and not the database file.
 - c Click **Finish**.
- 8 Click **OK** to close the Select Database dialog box. In the ODBC Microsoft Access Setup dialog box, click **OK**. The ODBC Data Source Administrator dialog box displays the `dbdemo` data source and any additional data sources that you added on the **User DSN** tab. Click **OK** to close the dialog box.

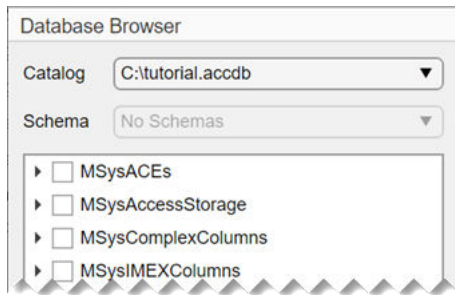
After you complete the data source setup, connect to the Access database using the Database Explorer app or the command line with the native ODBC connection.

Step 4. Connect using the Database Explorer app or the command line.

Connect to Access Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, leave the **Username** and **Password** boxes blank, and click **Connect**.

The app connects to the database and displays a list of its objects, such as tables, in the **Data Browser** pane. The data source tab named `dbdemo` appears to the right of the pane. The data source tab contains empty **SQL Query** and **Data Preview** panes.



- 3 Select tables in the **Data Browser** pane to query the database.
- 4 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to Access Using ODBC Driver and Command Line

- 1 Connect to the database with the ODBC data source name. For example, this code assumes that you are connecting to a data source named dbdemo with a blank user name and password.

```
conn = database('dbdemo', '', '');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

database | close

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

Microsoft SQL Server ODBC for Windows

This tutorial shows how to set up a data source and connect to a Microsoft SQL Server database using the Database Explorer app or the command line. The tutorial uses the Microsoft ODBC Driver 13.1 for SQL Server to connect to a Microsoft SQL Server 2016 Express database. Also, you can use these steps to configure Azure[®] Synapse SQL.

Step 1. Verify the driver installation.

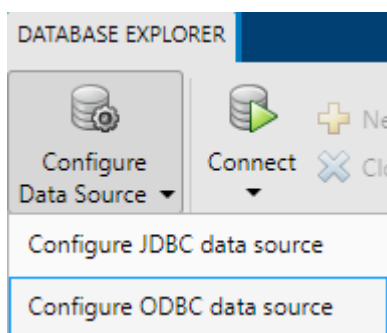
The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers. For information about the Microsoft ODBC Data Source Administrator, see Driver Installation.

Note Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of SQL Server. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see “Microsoft SQL Server JDBC for Windows” on page 2-29. For details about working with the 64-bit version of Windows, see Using Previous MATLAB Releases.

Step 2. Set up the data source using the Database Explorer app.

The Database Explorer app accesses the Microsoft ODBC Data Source Administrator automatically when you configure an ODBC data source. Alternatively, you can access the Microsoft ODBC Data Source Administrator using the `configureODBCDataSource` function.

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure ODBC data source**.

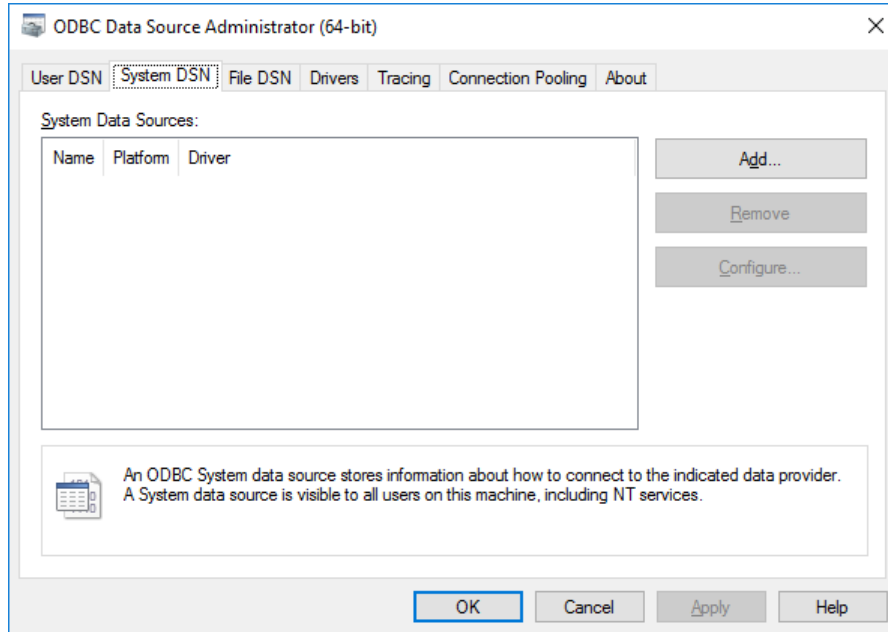


In the ODBC Data Source Administrator dialog box, you define the ODBC data source.

Tip When setting up an ODBC data source, you can specify a user data source name (DSN) or a system DSN. A user DSN is specific to the person logged into a machine. Only this person sees the data sources that are defined on the user DSN tab. A system DSN is not specific to the person logged into a machine. Any person who logs into the machine can see the data sources that are defined on the system DSN tab. Your ability to set up a user DSN or system DSN depends on the

database and ODBC driver you are using. For details, contact the database administrator or refer to the ODBC driver documentation.

- 3 Click the **System DSN** tab, and then click **Add**.



The Create New Data Source dialog box opens and displays a list of installed ODBC drivers.

- 4 Select SQL Server Native Client 11.0.

Note The name of the ODBC driver can vary.

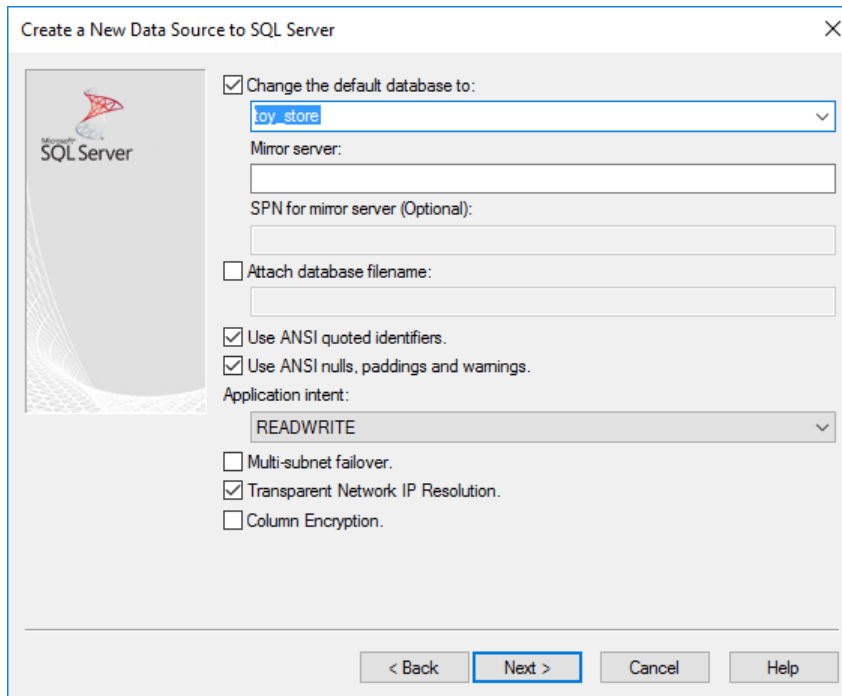
Click **Finish**.

- 5 In the Create a New Data Source to SQL Server dialog box, enter an appropriate name for the data source. You use this name to establish a connection to your database. Here, in the **Name** box, enter MS SQL Server as the data source name. In the **Description** box, enter Microsoft SQL Server as the description. From the **Server** list, select the database server for this data source to use. Consult your database administrator for the name of your database server. Click **Next**.
- 6 You can set up an ODBC data source with or without Windows authentication.

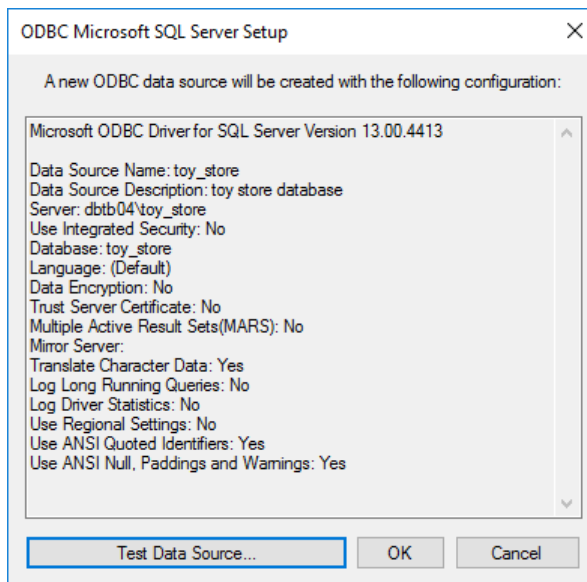
If you want to connect to SQL Server using Windows authentication, select **With Integrated Windows Authentication**. Then click **Next**.

Or, if you want to connect to SQL Server without Windows authentication, select **With SQL Server authentication using a login ID and password entered by the user**. Enter your user name in the **Login ID** box and your password in the **Password** box. Then click **Next**.

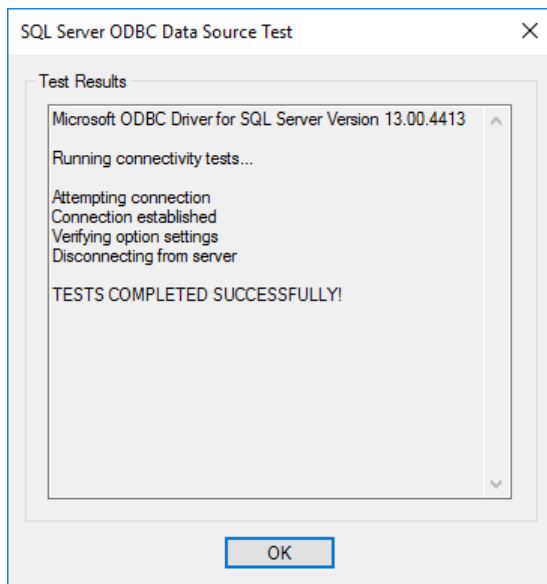
- 7 In the Create a New Data Source to SQL Server dialog box, select **Change the default database to** and enter the name of the default database on the database server for connection. Here, use the database `toy_store`. Then click **Next**.



- 8 Click **Finish** to accept the default settings.
- 9 In the ODBC Microsoft SQL Server Setup dialog box, test your connection by clicking **Test Data Source**.



- 10 If the connection succeeds, the SQL Server ODBC Data Source Test dialog box opens and displays a message indicating the tests completed successfully. Click **OK** to close this dialog box. Click **OK** to close the ODBC Microsoft SQL Server Setup dialog box.



- 11 The ODBC Data Source Administrator dialog box shows the new data source under System Data Sources on the **System DSN** tab. Click **OK** to close the ODBC Data Source Administrator dialog box.

After you complete the data source setup, connect to the SQL Server database using the Database Explorer app or the command line with the native ODBC connection.

Step 3. Connect using the Database Explorer app or the command line.

Connect to SQL Server Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 If you create a connection with Windows authentication, leave the **Username** and **Password** boxes blank in the connection dialog box, and click **Connect**. Otherwise, enter a user name and password, and click **Connect**.

The Catalog and Schema dialog box opens.

- 3 Select the catalog and schema from the **Catalog** and **Schema** lists. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to SQL Server Using ODBC Driver and Command Line

- 1 To connect with Windows authentication, connect to the database with the authenticated ODBC data source name and a blank user name and password. For example, this code assumes that you are connecting to a data source named MS SQL Server Auth.

```
conn = database('MS SQL Server Auth', '', '');
```

Or, to connect without Windows authentication, connect to the database with the ODBC data source name. For example, this code assumes that you are connecting to a data source named MS SQL Server with the user name `username` and the password `pwd`.

```
conn = database('MS SQL Server', 'username', 'pwd');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

database | close

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

Microsoft SQL Server ODBC for Windows DSN-Less Connection

This tutorial shows how to verify your driver installation and connect to a Microsoft SQL Server database using a DSN-less connection string at the command line. (DSN is a data source name.) The tutorial uses the Microsoft ODBC Driver 13.1 for SQL Server to connect to a Microsoft SQL Server 2016 Express database on the Windows platform.

Step 1. Verify the driver installation.

The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers.

Note Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of SQL Server. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see “Microsoft SQL Server JDBC for Windows” on page 2-29. For details about working with the 64-bit version of Windows, see Using Previous MATLAB Releases.

Step 2. Connect using the DSN-less connection string and command line.

- 1 Connect to the database using the DSN-less connection string with the `odbc` function. For example, this code assumes that you are connecting to the local database server, port number 1433, database name `toystore_doc`, user name `username`, and password `pwd`.

```
dsnless = strcat("Driver={SQL Server Native Client 13.1}; Server=localhost\toystore_doc", ...  
    "Port=1433; Database=toystore_doc; UID=username; PWD=pwd");  
conn = odbc(dsnless);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

`odbc` | `close`

Related Examples

- “Configure Driver and Data Source” on page 2-14
- “Microsoft SQL Server ODBC for Windows” on page 2-23
- “Microsoft SQL Server ODBC for Linux DSN-Less Connection” on page 2-79
- “Database Connection Error Messages” on page 3-7

Microsoft SQL Server JDBC for Windows

This tutorial shows how to set up a data source and connect to a Microsoft SQL Server database using the Database Explorer app or the command line. This tutorial uses the Microsoft JDBC Driver 4.0 for Microsoft SQL Server to connect to a Microsoft SQL Server 2016 Express database.

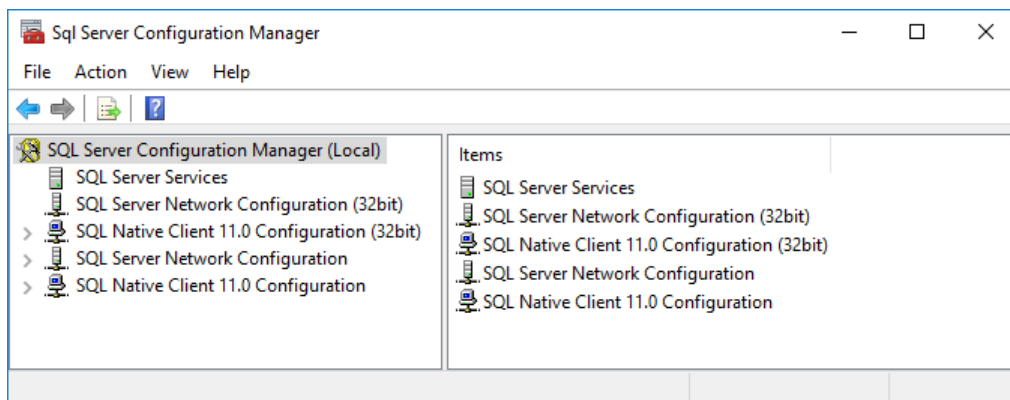
Step 1. Verify the driver installation.

If the JDBC driver for Microsoft SQL Server is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

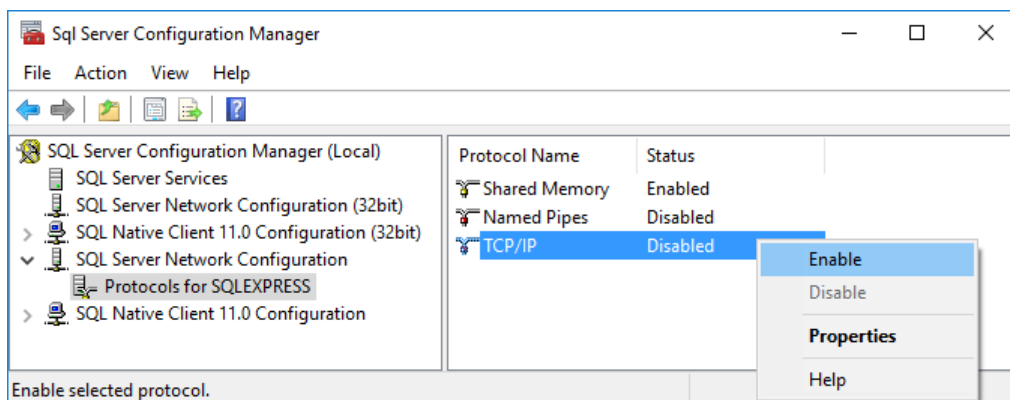
Step 2. Verify the port number.

Complete the following steps on the machine where SQL Server is installed to find your port number for database connection. If you experience connection issues with the port number that you find, contact your database administrator.

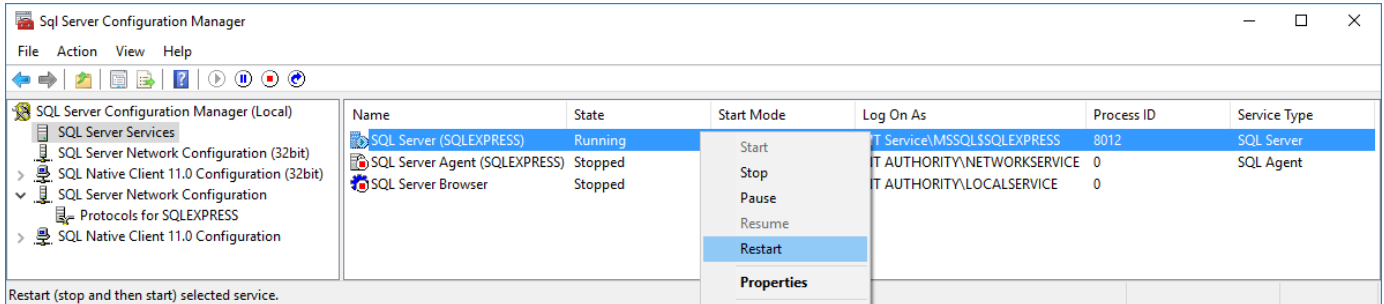
- 1 On the machine where your SQL Server database is installed, click **Start**. Select your Microsoft SQL Server version folder and click **SQL Server Configuration Manager**.



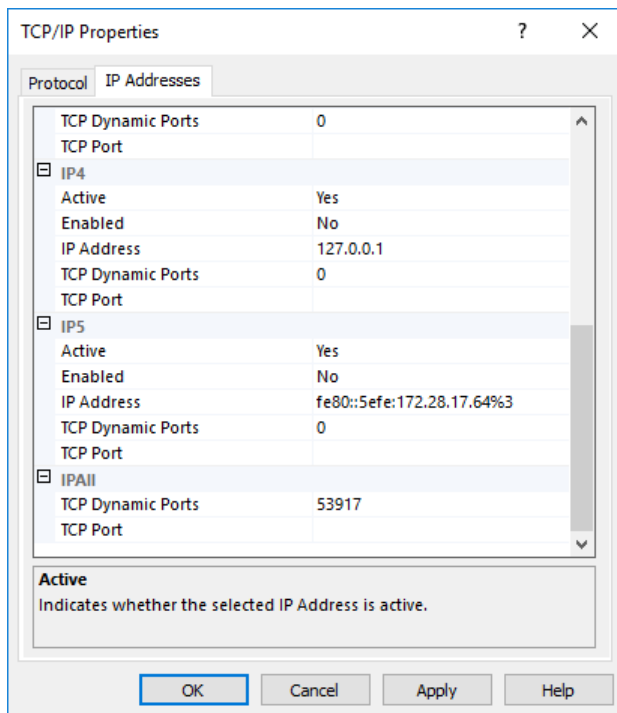
- 2 On the left of the Sql Server Configuration Manager window, click **SQL Server Network Configuration**. Double-click **Protocols for SQLEXPRESS**.
- 3 See if TCP/IP is enabled. If so, skip steps 4 and 5.
- 4 If TCP/IP is disabled, right-click **TCP/IP** and select **Enable**.



- To finish the process of enabling the TCP/IP protocol, restart the server. On the left side of the window, click **SQL Server Services**. Right-click **SQL Server (SQLEXPRESS)** and click **Restart**. The server restarts, enabling TCP/IP.



- Click **Protocols for SQLEXPRESS** and right-click **TCP/IP**. Select **Properties**.
- In the TCP/IP Properties dialog box, scroll to the bottom on the **IP Addresses** tab until you see the **IPAll** group. The number next to **TCP Dynamic Ports** is the port number. Use this port number in the JDBC Data Source Configuration dialog box when configuring a data source using the Database Explorer app. Or, enter this port number as an input argument of the **database** function at the command line. Here, the port number is 53917. If this number is 0 or if you want to configure your SQL Server database server to listen to a specific port, delete the entry in the **TCP Dynamic Ports** box. Then, enter another port number in the **TCP Port** box.



- Click **Apply** and click **OK** to close the TCP/IP Properties dialog box. Then, close the Sql Server Configuration Manager dialog box.

Step 3. Set up the operating system authentication.

Windows authentication enables you to connect to a database using your Windows user account. In this case, Windows performs user validation, and the database does not require a different user name and password. Windows authentication facilitates the maintenance of database access credentials. After you add the required libraries to the system path, the Microsoft SQL Server JDBC driver enables connectivity using Windows authentication. The following steps show how to add these libraries to the Java library path in MATLAB. For details about Java libraries, see “Java Class Path”.

- 1 Ensure that you have the latest Java driver library installed on your computer. To install the latest library, see [Driver Installation](#).
- 2 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 3 Close MATLAB.
- 4 Navigate to the folder from step 2, and create a file named `javajlibrarypath.txt` in the folder.
- 5 Open `javajlibrarypath.txt` and insert the path to the Java library file `sqljdbc_auth.dll`. This file is installed in the following location:

```
<installation>\sqljdbc_<version>\<language>\auth\<arch>
```

`<installation>` is the installation folder of the Microsoft SQL Server JDBC driver, `<version>` is the JDBC driver version, `<language>` is the JDBC driver language, and `<arch>` is the architecture.

Use the `x64` folder. In the entry, include the full path to the library file. Do not include the library file name. The following is an example of the path: `C:\DB_Drivers\sqljdbc_4.0\enu\auth\x64`. Save and close `javajlibrarypath.txt`.

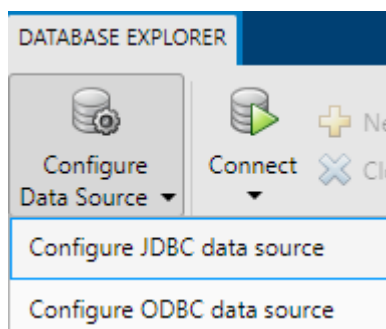
- 6 Restart MATLAB.

Step 4. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select Microsoft SQL Server.

JDBC Data Source Configuration

Data Source Details

Name: SQLite

Vendor: Microsoft SQL Server

Driver Location: [Browse]

Connection Parameters

Database: []

Server: localhost

Port Number: 1433

Authentication: Server

Connection Options

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Database** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.
- 7 To establish the data source with Windows authentication, set **Authentication** to Windows.

Or, to establish the data source without Windows authentication, set **Authentication** to Server.

- 8 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign + to specify additional driver-specific options.
- 9 Click **Test**. The Test Connection dialog box opens. If you are connecting without Windows authentication, then enter the user name and password for your database. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 10 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for an SQL Server database.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. To set the connection options with Windows authentication, use the 'AuthenticationType' name-value pair argument. For example, this code assumes that you are connecting to a JDBC data source named MSSQLServer, full path of the JDBC driver file C:\Drivers\sqljdbc4.jar, database name toystore_doc, database server dbtb04, port number 54317, and authentication type Windows.

```
opts = setoptions(opts, ...
    'DataSourceName', "MSSQLServer", ...
    'JDBCdriverLocation', "C:\Drivers\sqljdbc4.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...
    'PortNumber', 54317, 'AuthenticationType', "Windows");
```

To set the connection options without Windows authentication, omit the 'AuthenticationType' name-value pair argument.

- 3 For a data source with Windows authentication, test the database connection by leaving the user name and password blank.

```
username = "";
password = "";
status = testConnection(opts, username, password);
```

To test without Windows authentication, specify a user name and password.

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the SQL Server database using the Database Explorer app or the JDBC driver and command line.

Step 5. Connect using the Database Explorer app or the command line.

Connect to SQL Server Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 If you create a connection with Windows authentication, leave the **Username** and **Password** boxes blank in the connection dialog box, and click **Connect**. Otherwise, enter a user name and password, and click **Connect**.

The Catalog and Schema dialog box opens.

- 3 Select the catalog and schema from the **Catalog** and **Schema** lists. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to SQL Server Using JDBC Driver and Command Line

- 1 To connect with Windows authentication, use the configured JDBC data source and specify a blank user name and password. For example, this code assumes that you are connecting to a JDBC data source named `MSSQLServerAuth`.

```
datasource = "MSSQLServerAuth";
username = "";
password = "";
conn = database(datasource,username,password);
```

Or, to connect without Windows authentication, use the configured JDBC data source and specify the user name `username` and the password `pwd`. For example, this code assumes that you are connecting to a JDBC data source named `MSSQLServer`.

```
datasource = "MSSQLServer";
username = "username";
password = "pwd";
conn = database(datasource,username,password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps
Database Explorer

Functions

database | close | databaseConnectionOptions | saveAsDataSource | setoptions | testConnection

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

Oracle ODBC for Windows

This tutorial shows how to set up a data source and connect to an Oracle database using the Database Explorer app or the command line. This tutorial uses the OraClient11g_home1 ODBC driver to connect to an Oracle 11g Enterprise Edition database.

Step 1. Verify the driver installation.

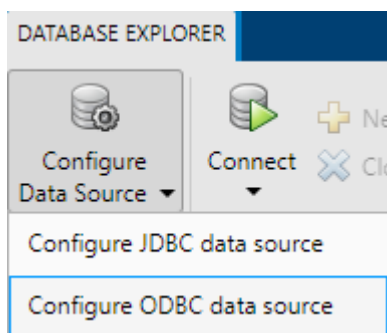
The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers. For information about the Microsoft ODBC Data Source Administrator, see Driver Installation.

Note: Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of Oracle. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see “Oracle JDBC for Windows” on page 2-41. For details about working with the 64-bit version of Windows, see Using Previous MATLAB Releases.

Step 2. Set up the data source using the Database Explorer app.

The Database Explorer app accesses the Microsoft ODBC Data Source Administrator automatically when you configure an ODBC data source. Alternatively, you can access the Microsoft ODBC Data Source Administrator using the `configureODBCDataSource` function.

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure ODBC data source**.

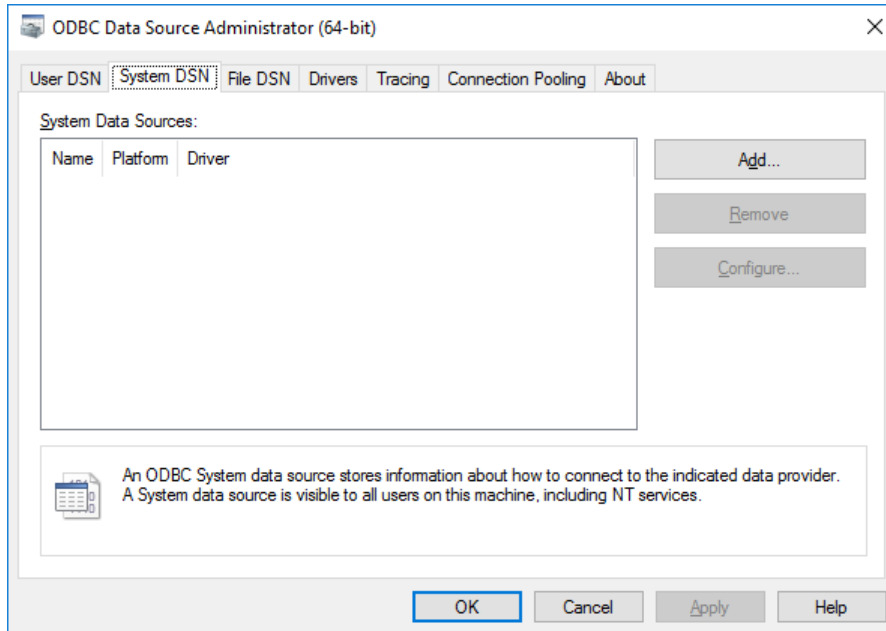


In the ODBC Data Source Administrator dialog box, you define the ODBC data source.

Tip When setting up an ODBC data source, you can specify a user data source name (DSN) or a system DSN. A user DSN is specific to the person logged into a machine. Only this person sees the data sources that are defined on the user DSN tab. A system DSN is not specific to the person logged into a machine. Any person who logs into the machine can see the data sources that are defined on the system DSN tab. Your ability to set up a user DSN or system DSN depends on the

database and ODBC driver you are using. For details, contact the database administrator or refer to the ODBC driver documentation.

- 3 Click the **System DSN** tab, and then click **Add**.



The Create New Data Source dialog box opens and displays a list of installed ODBC drivers.

- 4 Select the ODBC driver `Oracle in OraClient11g_home1`.

Note The name of the ODBC driver can vary.

Click **Finish**.

- 5 In the Oracle ODBC Driver Configuration dialog box, enter an appropriate name for the data source. You use this name to establish a connection to your database. Here, in the **Data Source Name** box, enter ORA as the data source name. In the **Description** box, enter a description for this data source, such as Oracle database. In the **TNS Service Name** box, enter the name of your database.
- 6 You can set up an ODBC data source with or without Windows authentication.

To establish the data source without Windows authentication, enter your user name in the **User ID** box. Or, to establish the data source with Windows authentication, leave this box blank. Leave the **Application**, **Oracle**, **Workarounds**, and **SQLServer Migration** tabs with the default settings.

- 7 Click **Test Connection** to test the connection to your database. The Oracle ODBC Driver Connect dialog box opens. If you are establishing the data source with Windows authentication, the Testing Connection dialog box opens.
- 8 Enter your password in the **Password** box. Your database name and user name are automatically entered in the **Service Name** and **User Name** boxes. Click **OK**. If your computer successfully connects to the database, the Testing Connection dialog box displays a message indicating the connection is successful. Click **OK**.
- 9 Click **OK** in the Oracle ODBC Driver Configuration dialog box. The ODBC Data Source Administrator dialog box shows the ODBC data source ORA.

After you complete the data source setup, connect to the Oracle database using the Database Explorer app or the command line with the native ODBC connection.

Step 3. Connect using the Database Explorer app or the command line.

Connect to Oracle Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 If you create a connection with Windows authentication, leave the **Username** and **Password** boxes blank in the connection dialog box, and click **Connect**. Otherwise, enter a user name and password, and click **Connect**.

The Catalog and Schema dialog box opens.

- 3 In the **Schema** list, select the schema. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to Oracle Using ODBC Driver and Command Line

- 1 To connect with Windows authentication, connect to the database with the authenticated ODBC data source name and a blank user name and password. For example, this code assumes that you are connecting to a data source named `Oracle_Auth`.

```
conn = database('Oracle_Auth', '', '');
```

Or, to connect without Windows authentication, connect to the database with the ODBC data source name. For example, this code assumes that you are connecting to a data source named `Oracle` with the user name `username` and the password `pwd`.

```
conn = database('Oracle', 'username', 'pwd');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

Oracle JDBC for Windows

This tutorial shows how to set up a data source and connect to an Oracle database using the Database Explorer app or the command line. This tutorial uses the Oracle Database 11g Release 2 (11.2.0.3) JDBC driver for use with JDK™ 1.6 to connect to an Oracle 11g Enterprise Edition Release 11.2.0.1.0 database.

Step 1. Verify the driver installation.

If the JDBC driver for Oracle is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the operating system authentication.

Windows authentication enables you to connect to a database using your system or network user name and password. In this case, the database does not require a different user name and password. Windows authentication facilitates connecting to the database and maintaining database access credentials. After you add the required libraries to the system path, the Oracle JDBC driver enables connectivity using Windows authentication. The following steps show how to add these libraries to the Java library path in MATLAB. For details about Java libraries, see “Java Class Path”.

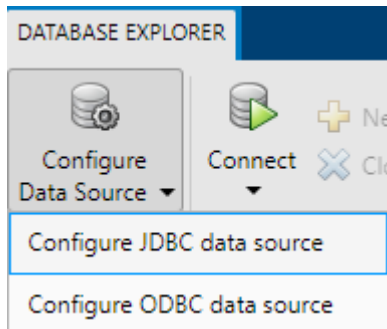
- 1 Ensure that you have the latest Oracle OCI libraries installed on your computer. To install the latest library, see Driver Installation.
- 2 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 3 Close MATLAB.
- 4 Navigate to the folder from step 2, and create a file named `javablibrarypath.txt` in the folder.
- 5 Open `javablibrarypath.txt` and insert the path to the Oracle OCI libraries. The entry must include the full path to the library files. The entry must not contain the library file names. The following is an example of the path: `C:\DB_Libraries\instantclient_11_2`. Save and close `javablibrarypath.txt`.
- 6 In the environment variables of the advanced system settings, add the Oracle OCI library full path to the Windows Path environment variable.
- 7 Restart MATLAB.

Step 3. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter the name of your data source. (This example uses a data source named ORA.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select Oracle.

JDBC Data Source Configuration

Data Source Details

Name:

Vendor:

- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL

Driver Location: ...

Connection Parameters

Database:

Server:

Port Number:

Driver Type:

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

Note To use the full entry from your `tnsnames.ora` file, select **Other** instead and enter the full entry in the resulting **URL** box. Then, enter the full path to the JDBC driver file in the **Driver Location** box and the name of the driver in the resulting **Driver** box. Save the JDBC data source.

For details about these steps, see “Other ODBC-Compliant or JDBC-Compliant Databases” on page 2-118.

- 5** In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6** In the **Database** box, enter the name of your database.

The name can be the service name or the Oracle system identifier (SID), depending on your specific Oracle database setup. For details, see your `tnsnames.ora` file, which is often stored in `<ORACLE_HOME>\NETWORK\ADMIN`, where `<ORACLE_HOME>` is the folder containing the installed database or Oracle client.

- 7** In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.
- 8** To establish the data source with Windows authentication, set **Driver Type** to `oci`.

Or, to establish the data source without Windows authentication, set **Driver Type** to `thin`.

- 9** Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign **+** to specify additional driver-specific options.
- 10** Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database. If you are connecting with Windows authentication, then leave these boxes blank. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 11** Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1** Create a JDBC data source for an Oracle database.

```
vendor = "Oracle";  
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2** Set the JDBC connection options. To set the connection options with Windows authentication, use the 'DriverType' name-value pair argument. For example, this code assumes that you are connecting to a JDBC data source named `ORA`, full path of the JDBC driver file `C:\Drivers\ojdbc7.jar`, database name `toystore_doc`, database server `dbtb05`, port number `1521`, and driver type `oci` (for Windows authentication).

```
opts = setoptions(opts, ...  
    'DataSourceName', "ORA", ...  
    'JDBCDriverLocation', "C:\Drivers\ojdbc7.jar", ...  
    'DatabaseName', "toystore_doc", 'Server', "dbtb05", ...  
    'PortNumber', 1521, 'DriverType', "oci");
```

To set the connection options without Windows authentication, omit the 'DriverType' name-value pair argument.

- 3** For a data source with Windows authentication, test the database connection by leaving the user name and password blank.

```
username = "";  
password = "";  
status = testConnection(opts, username, password);
```


To test without Windows authentication, specify a user name and password.

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the Oracle database using the Database Explorer app or the JDBC driver and command line.

Step 4. Connect using the Database Explorer app or the command line.

Connect to Oracle Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 If you create a connection with Windows authentication, leave the **Username** and **Password** boxes blank in the connection dialog box, and click **Connect**. Otherwise, enter a user name and password, and click **Connect**.

The Catalog and Schema dialog box opens.

- 3 In the **Schema** list, select the schema. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to Oracle Using JDBC Driver and Command Line

- 1 To connect with Windows authentication, use the configured JDBC data source and specify a blank user name and password. For example, this code assumes that you are connecting to a JDBC data source named `ORA_Auth`.

```
datasource = "ORA_Auth";
username = "";
password = "";
conn = database(datasource, username, password);
```

Or, to connect without Windows authentication, use the configured JDBC data source and specify the user name `username` and the password `pwd`. For example, this code assumes that you are connecting to a JDBC data source named `ORA`.

```
datasource = "ORA";
username = "username";
password = "pwd";
conn = database(datasource, username, password);
```

If you have trouble using the `database` function, use the full entry from your `tnsnames.ora` file in the URL string as one consecutive line. Leave the first argument blank. For example, this code assumes that the value of the 'URL' name-value pair argument is set to the specified `tnsnames.ora` file entry for an Oracle database.

```
conn = database('','username','pwd', ...  
    'Vendor','Oracle', ...  
    'URL',[ 'jdbc:oracle:thin:@(DESCRIPTION = ' ...  
    '(ADDRESS = (PROTOCOL = TCP)(HOST = sname)' ...  
    '(PORT = 123456)) (CONNECT_DATA = ' ...  
    '(SERVER = DEDICATED) (SERVICE_NAME = dbname) ) ']);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `saveAsDataSource` | `setoptions` | `testConnection`

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

MySQL ODBC for Windows

This tutorial shows how to set up a data source and connect to a MySQL database using the Database Explorer app or the command line. The tutorial uses a MySQL ODBC 5.3 Driver to connect to the MySQL database.

Step 1. Verify the driver installation.

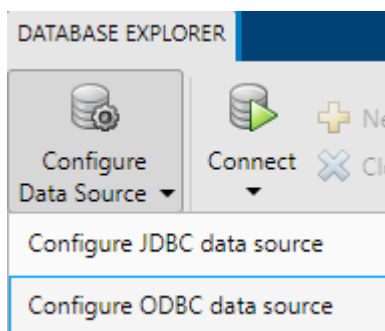
The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers. For information about the Microsoft ODBC Data Source Administrator, see Driver Installation.

Note: Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of MySQL. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see “MySQL JDBC for Windows” on page 2-52. For details about working with the 64-bit version of Windows, see Using Previous MATLAB Releases.

Step 2. Set up the data source using the Database Explorer app.

The Database Explorer app accesses the Microsoft ODBC Data Source Administrator automatically when you configure an ODBC data source. Alternatively, you can access the Microsoft ODBC Data Source Administrator using the `configureODBCDataSource` function.

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure ODBC data source**.

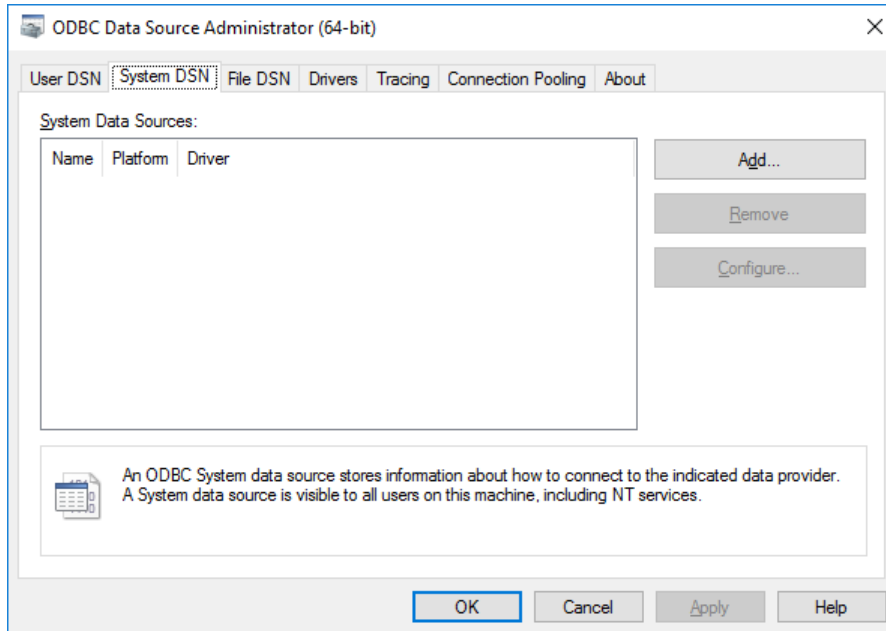


In the ODBC Data Source Administrator dialog box, you define the ODBC data source.

Tip When setting up an ODBC data source, you can specify a user data source name (DSN) or a system DSN. A user DSN is specific to the person logged into a machine. Only this person sees the data sources that are defined on the user DSN tab. A system DSN is not specific to the person logged into a machine. Any person who logs into the machine can see the data sources that are defined on the system DSN tab. Your ability to set up a user DSN or system DSN depends on the

database and ODBC driver you are using. For details, contact the database administrator or refer to the ODBC driver documentation.

- 3 Click the **System DSN** tab, and then click **Add**.



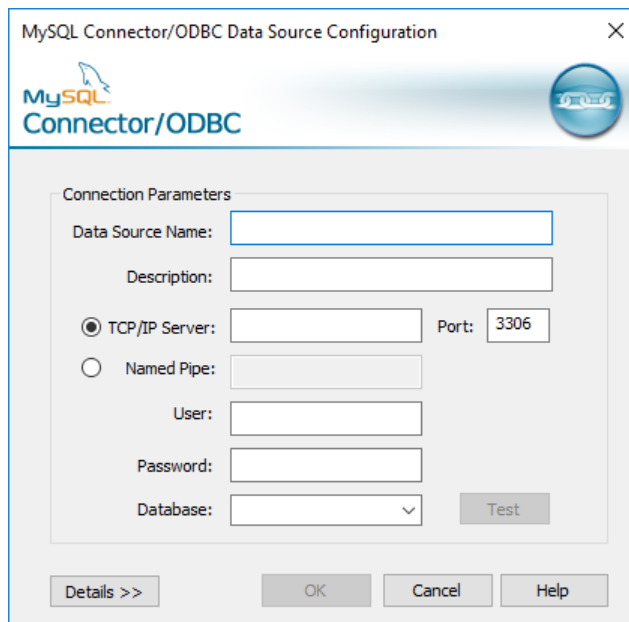
The Create New Data Source dialog box opens and displays a list of installed ODBC drivers.

- 4 Select the ODBC driver **MySQL ODBC 5.2a Driver**.

Note The name of the ODBC driver can vary.

Click **Finish**.

- 5 In the MySQL Connector/ODBC Data Source Configuration dialog box, fill out the boxes.



- In the **Data Source Name** box, enter an appropriate name for the data source, such as MySQL. You use this name to establish a connection to your database.
 - In the **Description** box, enter a description for this data source, such as MySQL database.
 - In the **TCP/IP Server** box, enter the name of your database server. Consult your database administrator for the name of your database server.
 - In the **Port** box, enter the port number. The default port number is 3306.
 - In the **User** box, enter your user name.
 - In the **Password** box, enter your password.
 - In the **Database** box, enter the name of your database.
- 6 Click **Test** to test the connection to your database. If your computer successfully connects to the database, the Test Result dialog box opens and displays a message indicating the connection is successful.
 - 7 Click **OK** in the MySQL Connector/ODBC Data Source Configuration dialog box. The ODBC Data Source Administrator dialog box shows the ODBC data source MySQL.

After you complete the data source setup, connect to the MySQL database using the Database Explorer app or the command line with the native ODBC connection.

Step 3. Connect using the Database Explorer app or the command line.

Connect to MySQL Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 In the **Catalog** list, select the catalog. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to MySQL Using ODBC Driver and Command Line

- 1 Connect to the database with the ODBC data source name. For example, this code assumes that you are connecting to a data source named MySQL with the user name `username` and the password `pwd`.

```
conn = database('MySQL', 'username', 'pwd');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

database | close

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

MySQL ODBC for Windows DSN-Less Connection

This tutorial shows how to verify your driver installation and connect to a MySQL database using a DSN-less connection string at the command line. (DSN is a data source name.) The tutorial uses a MySQL ODBC 5.3 Driver to connect to the MySQL database on the Windows platform.

Step 1. Verify the driver installation.

The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers.

Step 2. Connect using the DSN-less connection string and command line.

- 1 Connect to the database using the DSN-less connection string with the `odbc` function. For example, this code assumes that you are connecting to the local database server, database name `toystore_doc`, user name `username`, and password `pwd`.

```
dsnless = strcat("Driver={MySQL ODBC 5.3 Ansi Driver}; Server=localhost; ", ...  
                "Database=toystore_doc; UID=username; PWD=pwd");  
conn = odbc(dsnless);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

`odbc` | `close`

Related Examples

- “Configure Driver and Data Source” on page 2-14
- “MySQL ODBC for Windows” on page 2-47
- “MySQL ODBC for Linux DSN-Less Connection” on page 2-95
- “Database Connection Error Messages” on page 3-7

MySQL JDBC for Windows

This tutorial shows how to set up a data source and connect to a MySQL database using the Database Explorer app or the command line. This tutorial uses the MySQL Connector/J 5.1.46 driver to connect to a MySQL Version 5.5.16 database.

Step 1. Verify the driver installation.

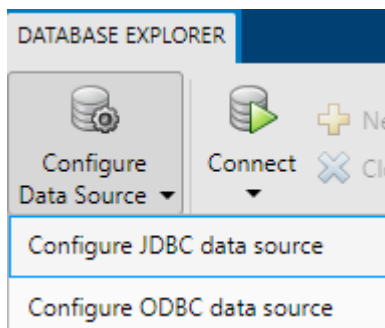
If the JDBC driver for MySQL is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter the name of your data source. (This example uses a data source named MySQL.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select MySQL.

JDBC Data Source Configuration

Data Source Details

Name:

Vendor: (Options: Microsoft SQL Server, MySQL, Oracle, PostgreSQL)

Driver Location: ...

Connection Parameters

Database:

Server:

Port Number:

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
 - 6 In the **Database** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.
 - 7 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign + to specify additional driver-specific options.
 - 8 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.
- If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.
- 9 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for a MySQL database.

```
vendor = "MySQL";  
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named MySQL, full path of the JDBC driver file C:\Drivers\mysql-connector-java-5.1.34-bin.jar, database name toystore_doc, database server dbtb01, and port number 3306.

```
opts = setoptions(opts, ...  
    'DataSourceName', "MySQL", ...  
    'JDBCDriverLocation', "C:\Drivers\mysql-connector-java-5.1.34-bin.jar", ...  
    'DatabaseName', "toystore_doc", 'Server', "dbtb01", ...  
    'PortNumber', 3306);
```

- 3 Test the database connection by specifying the user name username and password pwd, or leave these arguments blank if your database does not require them.

```
username = "username";  
password = "pwd";  
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the MySQL database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to MySQL Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 In the **Catalog** list, select the catalog. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to MySQL Using JDBC Driver and Command Line

- 1 Connect to a MySQL database using the configured JDBC data source, user name `username`, and password `pwd`.

```
datasource = "MySQL";  
username = "username";  
password = "pwd";  
conn = database(datasource,username,password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `saveAsDataSource` | `setoptions` | `testConnection`

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

PostgreSQL ODBC for Windows

This tutorial shows how to set up a data source and connect to a PostgreSQL database using the Database Explorer app or the command line. This tutorial uses the PostgreSQL ANSI(x64) driver to connect to a PostgreSQL 9.2 database.

Step 1. Verify the driver installation.

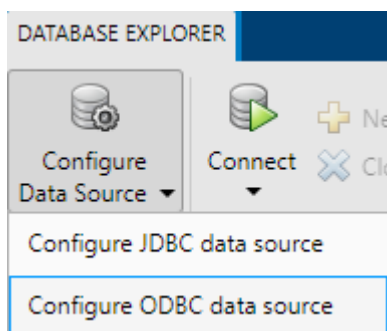
The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers. For information about the Microsoft ODBC Data Source Administrator, see Driver Installation.

Note: Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of PostgreSQL. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see “PostgreSQL JDBC for Windows” on page 2-61. For details about working with the 64-bit version of Windows, see Using Previous MATLAB Releases.

Step 2. Set up the data source using the Database Explorer app.

The Database Explorer app accesses the Microsoft ODBC Data Source Administrator automatically when you configure an ODBC data source. Alternatively, you can access the Microsoft ODBC Data Source Administrator using the `configureODBCDataSource` function.

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure ODBC data source**.

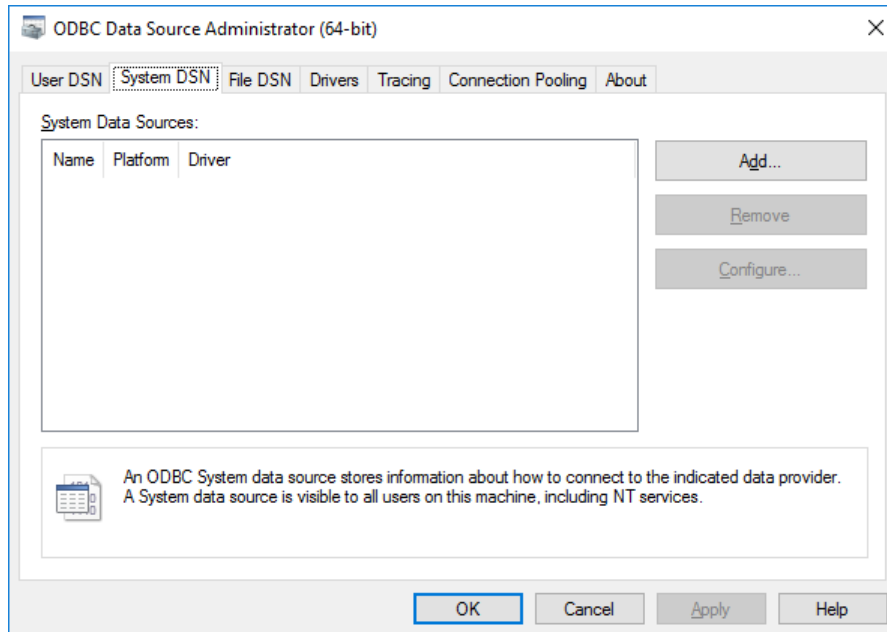


In the ODBC Data Source Administrator dialog box, you define the ODBC data source.

Tip When setting up an ODBC data source, you can specify a user data source name (DSN) or a system DSN. A user DSN is specific to the person logged into a machine. Only this person sees the data sources that are defined on the user DSN tab. A system DSN is not specific to the person logged into a machine. Any person who logs into the machine can see the data sources that are defined on the system DSN tab. Your ability to set up a user DSN or system DSN depends on the

database and ODBC driver you are using. For details, contact the database administrator or refer to the ODBC driver documentation.

- 3 Click the **System DSN** tab, and then click **Add**.



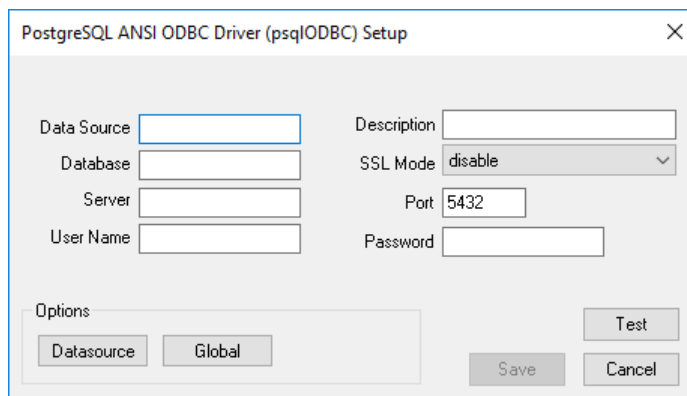
The Create New Data Source dialog box opens and displays a list of installed ODBC drivers.

- 4 Select the ODBC driver PostgreSQL ANSI (x64).

Note The name of the ODBC driver can vary.

Click **Finish**.

- 5 In the PostgreSQL ANSI ODBC Driver (psqlODBC) Setup dialog box, fill out the boxes.



- In the **Data Source** box, enter an appropriate name for the data source, such as PostgreSQL. You use this name to establish a connection to your database.
- In the **Description** box, enter a description for this data source, such as PostgreSQL database.

- In the **Database** box, enter the name of your database.
 - In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server.
 - In the **Port** box, enter the port number. The default port number is 5432.
 - In the **User Name** box, enter your user name.
 - In the **Password** box, enter your password.
- 6 Click **Test** to test the connection to your database. If your computer successfully connects to the database, the Connection Test dialog box opens and displays a message indicating the connection is successful.
 - 7 Click **Save** in the PostgreSQL ANSI ODBC Driver (psqlODBC) Setup dialog box. The ODBC Data Source Administrator dialog box shows the ODBC data source PostgreSQL30.

After you complete the data source setup, connect to the PostgreSQL database using the Database Explorer app or the command line with the native ODBC connection.

Step 3. Connect using the Database Explorer app or the command line.

Connect to PostgreSQL Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 Select the catalog and schema from the **Catalog** and **Schema** lists. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to PostgreSQL Using ODBC Driver and Command Line

- 1 Connect to the database with the ODBC data source name. For example, this code assumes that you are connecting to a data source named PostgreSQL with the user name username and the password pwd.

```
conn = database('PostgreSQL', 'username', 'pwd');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

database | close

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

PostgreSQL ODBC for Windows DSN-Less Connection

This tutorial shows how to verify your driver installation and connect to a PostgreSQL database using a DSN-less connection string at the command line. (DSN is a data source name.) The tutorial uses the JDBC4 PostgreSQL Driver, Version 8.4 to connect to a PostgreSQL 9.2 database on the Windows platform.

Step 1. Verify the driver installation.

The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers.

Step 2. Connect using the DSN-less connection string and command line.

- 1 Connect to the database using the DSN-less connection string with the `odbc` function. For example, this code assumes that you are connecting to the local database server, database name `toystore_doc`, user name `username`, and password `pwd`.

```
dsnless = strcat("Driver={PostgreSQL ANSI(x64)}; Server=localhost; ", ...  
    "Database=toystore_doc; UID=username; PWD=pwd");  
conn = odbc(dsnless);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

`odbc` | `close`

Related Examples

- “Configure Driver and Data Source” on page 2-14
- “PostgreSQL ODBC for Windows” on page 2-56
- “PostgreSQL ODBC for Linux” on page 2-104
- “Database Connection Error Messages” on page 3-7

PostgreSQL JDBC for Windows

This tutorial shows how to set up a data source and connect to a PostgreSQL database using the Database Explorer app or the command line. This tutorial uses the JDBC4 PostgreSQL Driver, Version 8.4 to connect to a PostgreSQL 9.2 database.

Step 1. Verify the driver installation.

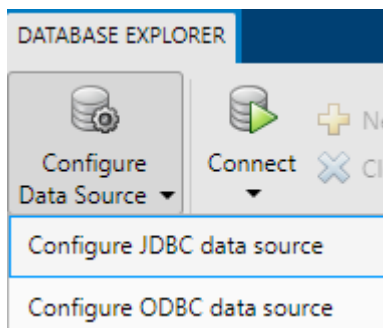
If the JDBC driver for PostgreSQL is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

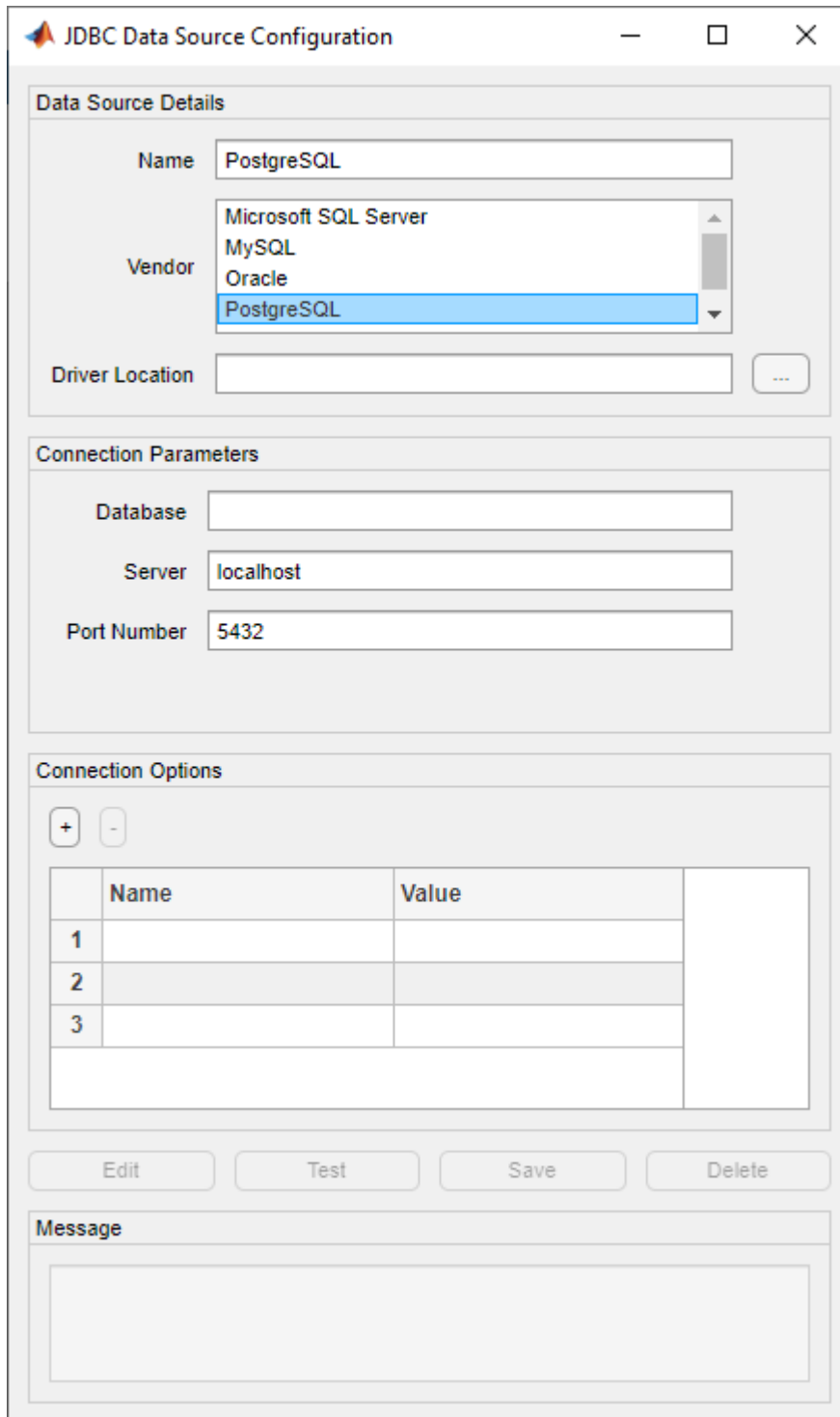
Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. (This example uses a data source named PostgreSQL.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select PostgreSQL.



- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Database** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.

- 7 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign + to specify additional driver-specific options.
- 8 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.
- 9 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for a PostgreSQL database.

```
vendor = "PostgreSQL";
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named PostgreSQL, full path of the JDBC driver file C:\Drivers\postgresql-8.4-702.jdbc4.jar, database name toystore_doc, database server dbtb00, and port number 5432.

```
opts = setoptions(opts, ...
    'DataSourceName', "PostgreSQL", ...
    'JDBCDriverLocation', "C:\Drivers\postgresql-8.4-702.jdbc4.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb00", ...
    'PortNumber', 5432);
```

- 3 Test the database connection by specifying the user name username and password pwd, or leave these arguments blank if your database does not require them.

```
username = "username";
password = "pwd";
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the PostgreSQL database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to PostgreSQL Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 Select the catalog and schema from the **Catalog** and **Schema** lists. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to PostgreSQL Using JDBC Driver and Command Line

- 1 Connect to a PostgreSQL database using the configured JDBC data source, user name `username`, and password `pwd`.

```
datasource = "PostgreSQL";  
username = "username";  
password = "pwd";  
conn = database(datasource, username, password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `saveAsDataSource` | `setoptions` | `testConnection`

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

SQLite JDBC for Windows

This tutorial shows how to set up a data source and connect to an SQLite database using the Database Explorer app or the command line. This tutorial uses the SQLite JDBC 3.7.2 Driver to connect to an SQLite Version 3.7.17 database.

Step 1. Verify the driver installation.

If the JDBC driver for SQLite is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

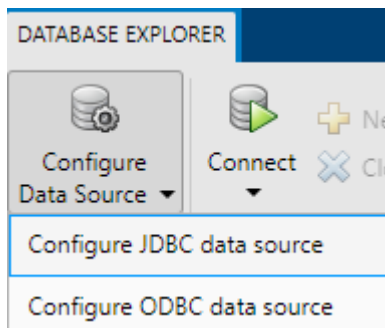
If you do not want to install a driver and want to store relational data quickly, you can use the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

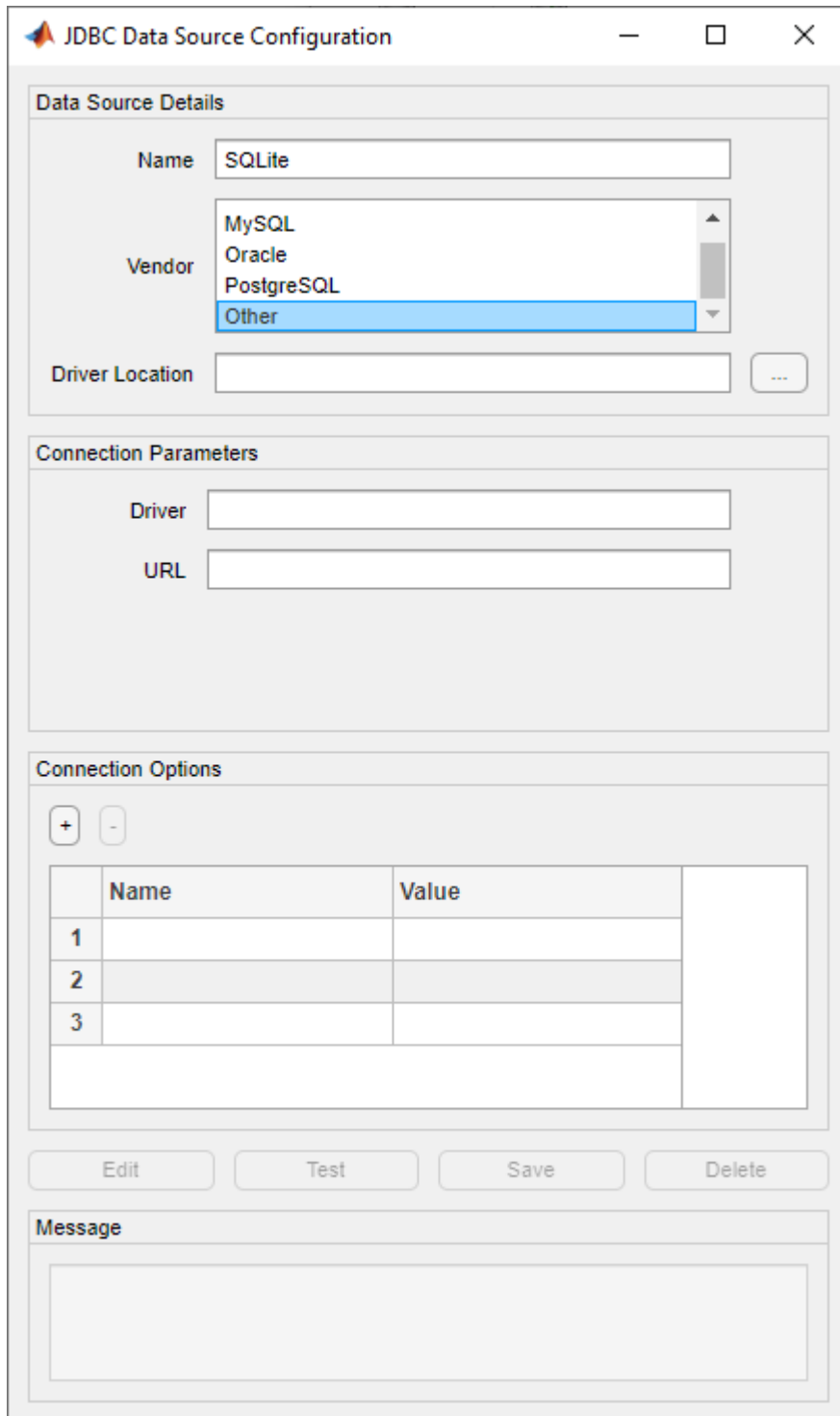
Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. (This example uses a data source named SQLite.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select **Other**.



- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Driver** box, enter the SQLite driver Java class object. Here, use `org.sqlite.JDBC`.

Note Your entries for **Driver** and **URL** can vary depending on the type and version of the JDBC driver and your database. For details, see the JDBC driver documentation for your database.

- 7 Connect to the SQLite database by creating a URL string using the format `jdbc:subprotocol:subname`. The `jdbc` part of this string remains constant for any JDBC driver. `subprotocol` is a database type, in this case, `sqlite`. For SQLite, `subname` contains the location of the database. For example, your URL string is `jdbc:sqlite:dbpath`, where `dbpath` is the full path to your SQLite database on your computer. Enter your string in the **URL** box and press **Enter**.
- 8 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign **+** to specify additional driver-specific options.
- 9 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 10 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for an SQLite database.

```
vendor = "Other";
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named SQLite, full path of the SQLite driver location `C:\Drivers\sqlite-jdbc-3.8.11.2.jar`, SQLite driver Java class object `org.sqlite.JDBC`, and URL string `jdbc:sqlite:C:\Databases\sqlite.db`.

```
opts = setoptions(opts, ...
    'DataSourceName', "SQLite", ...
    'JDBCDriverLocation', "C:\Drivers\sqlite-jdbc-3.8.11.2.jar", ...
    'Driver', "org.sqlite.JDBC", ...
    'URL', "jdbc:sqlite:C:\Databases\sqlite.db");
```

- 3 Test the database connection by specifying the user name `username` and password `pwd`, or leave these arguments blank if your database does not require them.

```
username = "username";
password = "pwd";
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the SQLite database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to SQLite Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 3 Select tables in the **Data Browser** pane to query the database.
- 4 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to SQLite Using JDBC Driver and Command Line

- 1 Connect to an SQLite database using the configured JDBC data source, user name `username`, and password `pwd`.

```
datasource = "SQLite";  
username = "username";  
password = "pwd";  
conn = database(datasource, username, password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `saveAsDataSource` | `setoptions` | `testConnection`

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14

- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

Microsoft SQL Server JDBC for macOS

This tutorial shows how to set up a data source and connect to a Microsoft SQL Server database using the Database Explorer app or the command line. This tutorial uses the Microsoft JDBC Driver 4.0 for Microsoft SQL Server to connect to a Microsoft SQL Server 2016 Express database.

Step 1. Verify the driver installation.

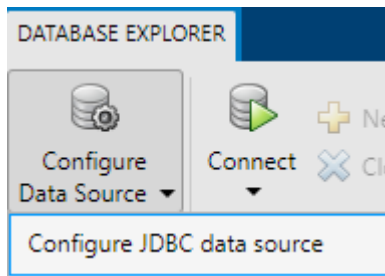
If the JDBC driver for SQL Server is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select Microsoft SQL Server.

Data Source Details

Name

Vendor

Driver Location ...

Connection Parameters

Database

Server

Port Number

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Database** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.
- 7 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign + to specify additional driver-specific options.
- 8 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 9 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for an SQL Server database.

```
vendor = "Microsoft SQL Server";  
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named MSSQLServer, full path of the JDBC driver file /home/user/DB_Drivers/sqljdbc4.jar, database name toystore_doc, database server dbtb04, and port number 54317.

```
opts = setoptions(opts, ...  
    'DataSourceName', "MSSQLServer", ...  
    'JDBCDriverLocation', "/home/user/DB_Drivers/sqljdbc4.jar", ...  
    'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...  
    'PortNumber', 54317);
```

- 3 Test the database connection by specifying the user name username and password pwd, or leave these arguments blank if your database does not require them.

```
username = "username";  
password = "pwd";  
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the SQL Server database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to SQL Server Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 Select the catalog and schema from the **Catalog** and **Schema** lists. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to SQL Server Using JDBC Driver and Command Line

- 1 Connect to an SQL Server database using the configured JDBC data source, user name username, and password pwd. For example, this code assumes that you are connecting to a JDBC data source named MSSQLServer.

```
datasource = "MSSQLServer";  
username = "username";  
password = "pwd";  
conn = database(datasource, username, password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

database | close | databaseConnectionOptions | saveAsDataSource | setoptions | testConnection

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

Microsoft SQL Server JDBC for Linux

This tutorial shows how to set up a data source and connect to a Microsoft SQL Server database using the Database Explorer app or the command line. This tutorial uses the Microsoft JDBC Driver 4.0 for Microsoft SQL Server to connect to a Microsoft SQL Server 2016 Express database.

Step 1. Verify the driver installation.

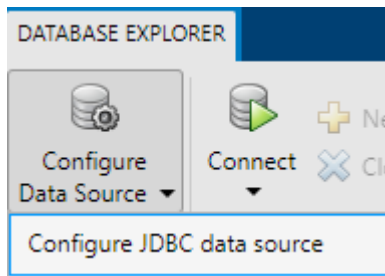
If the JDBC driver for Microsoft SQL Server is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select Microsoft SQL Server.

JDBC Data Source Configuration

Data Source Details

Name

Vendor Microsoft SQL Server
 MySQL
 Oracle
 PostgreSQL

Driver Location ...

Connection Parameters

Database

Server

Port Number

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Database** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.

- 7 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign + to specify additional driver-specific options.
- 8 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 9 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for an SQL Server database.

```
vendor = "Microsoft SQL Server";  
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named MSSQLServer, full path of the JDBC driver file /home/user/DB_Drivers/sqljdbc4.jar, database name toystore_doc, database server dbtb04, and port number 54317.

```
opts = setoptions(opts, ...  
    'DataSourceName', "MSSQLServer", ...  
    'JDBCDriverLocation', "/home/user/DB_Drivers/sqljdbc4.jar", ...  
    'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...  
    'PortNumber', 54317);
```

- 3 Test the database connection by specifying the user name username and password pwd, or leave these arguments blank if your database does not require them.

```
username = "username";  
password = "pwd";  
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the SQL Server database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to SQL Server Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 Select the catalog and schema from the **Catalog** and **Schema** lists. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to SQL Server Using JDBC Driver and Command Line

- 1 Connect to an SQL Server database using the configured JDBC data source, user name username, and password pwd. For example, this code assumes that you are connecting to a JDBC data source named MSSQLServer.

```
datasource = "MSSQLServer";
username = "username";
password = "pwd";
conn = database(datasource, username, password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

database | close | databaseConnectionOptions | saveAsDataSource | setoptions | testConnection

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

Microsoft SQL Server ODBC for Linux

This tutorial shows how to set up a data source and connect to a Microsoft SQL Server database using the command line. The tutorial uses the Microsoft ODBC Driver 13.1 for Microsoft SQL Server to connect to a Microsoft SQL Server 2016 Express database on the Linux platform.

Step 1. Verify the driver installation.

The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers.

If you need to install an ODBC driver, contact your database administrator or see [Install the Microsoft ODBC Driver for SQL Server \(Linux\)](#).

Step 2. Set up the data source.

Follow the instructions in [Connecting to SQL Server](#) to create a data source name (DSN).

Step 3. Connect using the command line.

- 1 Connect to the database using the configured DSN, user name `username`, and password `pwd` with the `odbc` function. For example, this code assumes that you are connecting to an ODBC data source `MSSQL`.

```
datasource = "MSSQL";  
username = "username";  
password = "pwd";  
conn = odbc(datasource,username,password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Functions

`odbc` | `close`

More About

- “Configure Driver and Data Source” on page 2-14
- “Microsoft SQL Server ODBC for Linux DSN-Less Connection” on page 2-79
- “Modify and Delete Data Sources” on page 4-17
- “Database Connection Error Messages” on page 3-7

Microsoft SQL Server ODBC for Linux DSN-Less Connection

This tutorial shows how to verify your driver installation and connect to a Microsoft SQL Server database using a DSN-less connection string at the command line. (DSN is a data source name.) The tutorial uses the Microsoft ODBC Driver 13.1 for SQL Server to connect to a Microsoft SQL Server 2016 Express database on the Linux platform.

Step 1. Verify the driver installation.

The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers.

If you need to install an ODBC driver, contact your database administrator or see [Install the Microsoft ODBC Driver for SQL Server \(Linux\)](#).

Step 2. Connect using the DSN-less connection string and command line.

- 1 Connect to the database using the DSN-less connection string with the `odbc` function. For example, this code assumes that you are connecting to the local database server, port number 1433, database name `toystore_doc`, user name `username`, and password `pwd`.

```
dsnless = strcat("Driver={ODBC Driver 13.1 for SQL Server}; Server=localhost,1433;", ...  
               "Database=toystore_doc; UID=username; PWD=pwd");  
conn = odbc(dsnless);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

[odbc](#) | [close](#)

Related Examples

- “Configure Driver and Data Source” on page 2-14
- “Microsoft SQL Server ODBC for Linux” on page 2-78
- “Database Connection Error Messages” on page 3-7

Oracle JDBC for macOS

This tutorial shows how to set up a data source and connect to an Oracle database using the Database Explorer app or the command line. This tutorial uses the Oracle Database 11g Release 2 (11.2.0.3) JDBC driver for use with JDK 1.6 to connect to an Oracle 11g Enterprise Edition Release 11.2.0.1.0 database.

Step 1. Verify the driver installation.

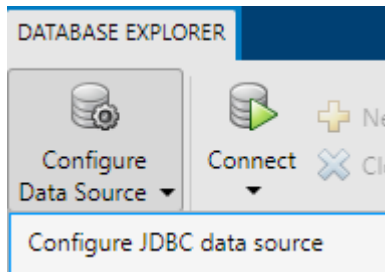
If the JDBC driver for Oracle is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. (This example uses a data source named `ORA`.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select `Oracle`.

Data Source Details

Name:

Vendor: (Options: Microsoft SQL Server, MySQL, Oracle, PostgreSQL)

Driver Location: ...

Connection Parameters

Database:

Server:

Port Number:

Driver Type: ▼

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

Note To use the full entry from your `tnsnames.ora` file, select **Other** instead and enter the full entry in the resulting **URL** box. Then, enter the full path to the JDBC driver file in the **Driver Location** box and the name of the driver in the resulting **Driver** box. Save the JDBC data source. For details about these steps, see “Other ODBC-Compliant or JDBC-Compliant Databases” on page 2-118.

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Database** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server.

The name can be the service name or the Oracle system identifier (SID), depending on your specific Oracle database setup. For details, see your `tnsnames.ora` file, which is often stored in `<ORACLE_HOME>/NETWORK/ADMIN`, where `<ORACLE_HOME>` is the folder containing the installed database or the Oracle client.

- 7 In the **Port Number** box, enter the port number. From the **Driver Type** list, select `thin` or `oci`. (Use `thin` as the default driver. Use `oci` if you installed an OCI driver.)
- 8 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign **+** to specify additional driver-specific options.
- 9 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 10 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for an Oracle database.

```
vendor = "Oracle";
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. To set the connection options with an OCI driver, use the 'DriverType' name-value pair argument. For example, this code assumes that you are connecting to a JDBC data source named `ORA`, full path of the JDBC driver file `/home/user/DB_Drivers/ojdbc7.jar`, database name `toystore_doc`, database server `dbtb05`, port number `1521`, and driver type `oci`.

```
opts = setoptions(opts, ...
    'DataSourceName', "ORA", ...
    'JDBCDriverLocation', "/home/user/DB_Drivers/ojdbc7.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb05", ...
    'PortNumber', 1521, 'DriverType', "oci");
```

To set the connection options without the OCI driver, omit the 'DriverType' name-value pair argument.

- 3 Test the database connection by specifying the user name `username` and password `pwd`, or leave these arguments blank if your database does not require them.

```
username = "username";
password = "pwd";
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the Oracle database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to Oracle Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 In the **Schema** list, select the schema. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to Oracle Using JDBC Driver and Command Line

- 1 Connect to an Oracle database using the configured JDBC data source, user name username, and password pwd.

```
datasource = "ORA";
username = "username";
password = "pwd";
conn = database(datasource, username, password);
```

If you have trouble using the `database` function, use the full entry from your `tnsnames.ora` file in the URL string as one consecutive line. Leave the first argument blank. For example, this code assumes that the value of the 'URL' name-value pair argument is set to the specified `tnsnames.ora` file entry for an Oracle database.

```
conn = database('','username','pwd', ...
    'Vendor','Oracle', ...
    'URL', ['jdbc:oracle:thin:@(DESCRIPTION = ' ...
    '(ADDRESS = (PROTOCOL = TCP)(HOST = sname)' ...
    '(PORT = 123456)) (CONNECT_DATA = ' ...
    '(SERVER = DEDICATED) (SERVICE_NAME = dbname) ) )']);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

database | close | databaseConnectionOptions | saveAsDataSource | setoptions | testConnection

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

Oracle JDBC for Linux

This tutorial shows how to set up a data source and connect to a Oracle database using the Database Explorer app or the command line. This tutorial uses the Oracle Database 11g Release 2 (11.2.0.3) JDBC driver for use with JDK 1.6 to connect to a Oracle 11g Enterprise Edition Release 11.2.0.1.0 database.

Step 1. Verify the driver installation.

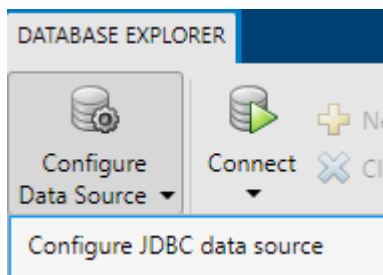
If the JDBC driver for Oracle is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. (This example uses a data source named `ORA`.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select `Oracle`.

JDBC Data Source Configuration

Data Source Details

Name:

Vendor:
 Microsoft SQL Server
 MySQL
Oracle
 PostgreSQL

Driver Location: ...

Connection Parameters

Database:

Server:

Port Number:

Driver Type:

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

Note To use the full entry from your `tnsnames.ora` file, select **Other** instead and enter the full entry in the resulting **URL** box. Then, enter the full path to the JDBC driver file in the **Driver Location** box and the name of the driver in the resulting **Driver** box. Save the JDBC data source. For details about these steps, see “Other ODBC-Compliant or JDBC-Compliant Databases” on page 2-118.

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Database** box, enter the name of your database.

The name can be the service name or the Oracle system identifier (SID), depending on your specific Oracle database setup. For details, see your `tnsnames.ora` file, which is often stored in `<ORACLE_HOME>/NETWORK/ADMIN`, where `<ORACLE_HOME>` is the folder containing the installed database or the Oracle client.

- 7 In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number. From the **Driver Type** list, select `thin` or `oci`. (Use `thin` as the default driver. Use `oci` if you installed an OCI driver.)
- 8 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign `+` to specify additional driver-specific options.
- 9 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 10 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for an Oracle database.

```
vendor = "Oracle";
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. To set the connection options with an OCI driver, use the 'DriverType' name-value pair argument. For example, this code assumes that you are connecting to a JDBC data source named `ORA`, full path of the JDBC driver file `/home/user/DB_Drivers/ojdbc7.jar`, database name `toystore_doc`, database server `dbtb05`, port number `1521`, and driver type `oci`.

```
opts = setoptions(opts, ...
    'DataSourceName', "ORA", ...
    'JDBCDriverLocation', "/home/user/DB_Drivers/ojdbc7.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb05", ...
    'PortNumber', 1521, 'DriverType', "oci");
```

To set the connection options without the OCI driver, omit the 'DriverType' name-value pair argument.

- 3 Test the database connection by specifying the user name `username` and password `pwd`, or leave these arguments blank if your database does not require them.

```
username = "username";
password = "pwd";
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the Oracle database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to Oracle Using Database Explorer App

- 1** On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2** In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3** In the **Schema** list, select the schema. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4** Select tables in the **Data Browser** pane to query the database.
- 5** Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to Oracle Using JDBC Driver and Command Line

- 1** Connect to an Oracle database using the configured JDBC data source, user name `username`, and password `pwd`.

```
datasource = "ORA";
username = "username";
password = "pwd";
conn = database(datasource,username,password);
```

If you have trouble using the `database` function, use the full entry from your `tnsnames.ora` file in the URL string as one consecutive line. Leave the first argument blank. For example, this code assumes that the value of the 'URL' name-value pair argument is set to the specified `tnsnames.ora` file entry for an Oracle database.

```
conn = database('','username','pwd', ...
    'Vendor','Oracle', ...
    'URL',[ 'jdbc:oracle:thin:@(DESCRIPTION = ' ...
    '(ADDRESS = (PROTOCOL = TCP)(HOST = sname)' ...
    '(PORT = 123456)) (CONNECT_DATA = ' ...
    '(SERVER = DEDICATED) (SERVICE_NAME = dbname) ) ) ]');
```

- 2** Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

database | close | databaseConnectionOptions | saveAsDataSource | setoptions | testConnection

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

MySQL JDBC for macOS

This tutorial shows how to set up a data source and connect to a MySQL database using the Database Explorer app or the command line. This tutorial uses the MySQL Connector/J 5.1.46 driver to connect to a MySQL Version 5.5.16 database.

Step 1. Verify the driver installation.

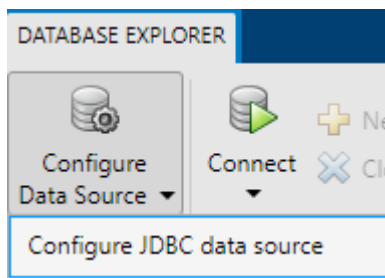
If the JDBC driver for MySQL is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. (This example uses a data source named MySQL.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select MySQL.

Data Source Details

Name:

Vendor: (Dropdown menu showing: Microsoft SQL Server, MySQL, Oracle, PostgreSQL)

Driver Location: ...

Connection Parameters

Database:

Server:

Port Number:

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Database** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.
- 7 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign **+** to specify additional driver-specific options.
- 8 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 9 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for a MySQL database.

```
vendor = "MySQL";  
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named MySQL, full path of the JDBC driver file /home/user/DB_Drivers/mysql-connector-java-5.1.17-bin.jar, database name toystore_doc, database server dbtb01, and port number 3306.

```
opts = setoptions(opts, ...  
    'DataSourceName', "MySQL", ...  
    'JDBCDriverLocation', "/home/user/DB_Drivers/mysql-connector-java-5.1.17-bin.jar", ...  
    'DatabaseName', "toystore_doc", 'Server', "dbtb01", ...  
    'PortNumber', 3306);
```

- 3 Test the database connection by specifying the user name username and password pwd, or leave these arguments blank if your database does not require them.

```
username = "username";  
password = "pwd";  
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the MySQL database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to MySQL Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 In the **Catalog** list, select the catalog. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to MySQL Using JDBC Driver and Command Line

- 1 Connect to a MySQL database using the configured JDBC data source, user name `username`, and password `pwd`.

```
datasource = "MySQL";  
username = "username";  
password = "pwd";  
conn = database(datasource, username, password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `saveAsDataSource` | `setoptions` | `testConnection`

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

MySQL ODBC for Linux

This tutorial shows how to set up a data source and connect to a MySQL database using the command line. The tutorial uses the MySQL Connector/J 5.1.46 driver to connect to a MySQL Version 5.5.16 database on the Linux platform.

Step 1. Verify the driver installation.

The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers.

If you need to install an ODBC driver, contact your database administrator or see *Installing Connector/ODBC on Unix-Like Systems*.

Step 2. Set up the data source.

Follow the instructions in *Configuring a Connector/ODBC DSN on Unix* to create a data source name (DSN).

Step 3. Connect using the command line.

- 1 Connect to the database using the configured DSN, user name `username`, and password `pwd` with the `odbc` function. For example, this code assumes that you are connecting to an ODBC data source `MySQL`.

```
datasource = "MySQL";  
username = "username";  
password = "pwd";  
conn = odbc(datasource,username,password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Functions

`odbc` | `close`

More About

- “Configure Driver and Data Source” on page 2-14
- “MySQL ODBC for Linux DSN-Less Connection” on page 2-95
- “Modify and Delete Data Sources” on page 4-17
- “Database Connection Error Messages” on page 3-7

MySQL ODBC for Linux DSN-Less Connection

This tutorial shows how to verify your driver installation and connect to a MySQL database using a DSN-less connection string at the command line. (DSN is a data source name.) The tutorial uses the MySQL Connector/J 5.3 driver to connect to a MySQL Version 5.5.16 database on the Linux platform.

Step 1. Verify the driver installation.

The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers.

If you need to install an ODBC driver, contact your database administrator or see [Installing Connector/ODBC on Unix-Like Systems](#).

Step 2. Connect using the DSN-less connection string and command line.

- 1 Connect to the database using the DSN-less connection string with the `odbc` function. For example, this code assumes that you are connecting to the local database server, database name `toystore_doc`, user name `username`, and password `pwd`.

```
dsnless = strcat("Driver={MySQL ODBC 5.3 Ansi Driver}; Server=localhost; ", ...  
               "Database=toystore_doc; UID=username; PWD=pwd");  
conn = odbc(dsnless);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

[odbc](#) | [close](#)

Related Examples

- “Configure Driver and Data Source” on page 2-14
- “MySQL ODBC for Linux” on page 2-94
- “Database Connection Error Messages” on page 3-7

MySQL JDBC for Linux

This tutorial shows how to set up a data source and connect to a MySQL database using the Database Explorer app or the command line. This tutorial uses the MySQL Connector/J 5.1.46 driver to connect to a MySQL Version 5.5.16 database.

Step 1. Verify the driver installation.

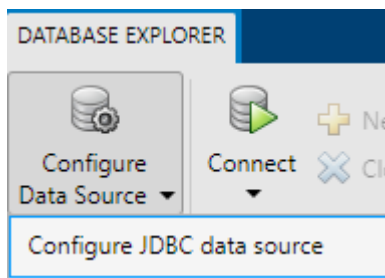
If the JDBC driver for MySQL is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. (This example uses a data source named MySQL.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select MySQL.

JDBC Data Source Configuration

Data Source Details

Name:

Vendor: (Options: Microsoft SQL Server, MySQL, Oracle, PostgreSQL)

Driver Location: ...

Connection Parameters

Database:

Server:

Port Number:

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Database** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.

- 7 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign **+** to specify additional driver-specific options.
- 8 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 9 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for a MySQL database.

```
vendor = "MySQL";  
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named MySQL, full path of the JDBC driver file /home/user/DB_Drivers/mysql-connector-java-5.1.17-bin.jar, database name toystore_doc, database server dbtb01, and port number 3306.

```
opts = setoptions(opts, ...  
    'DataSourceName', "MySQL", ...  
    'JDBCDriverLocation', "/home/user/DB_Drivers/mysql-connector-java-5.1.17-bin.jar", ...  
    'DatabaseName', "toystore_doc", 'Server', "dbtb01", ...  
    'PortNumber', 3306);
```

- 3 Test the database connection by specifying the user name username and password pwd, or leave these arguments blank if your database does not require them.

```
username = "username";  
password = "pwd";  
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the MySQL database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to MySQL Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 In the **Catalog** list, select the catalog. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to MySQL Using JDBC Driver and Command Line

- 1 Connect to a MySQL database using the configured JDBC data source, user name `username`, and password `pwd`.

```
datasource = "MySQL";
username = "username";
password = "pwd";
conn = database(datasource, username, password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `saveAsDataSource` | `setoptions` | `testConnection`

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

PostgreSQL JDBC for macOS

This tutorial shows how to set up a data source and connect to a PostgreSQL database using the Database Explorer app or the command line. This tutorial uses the JDBC4 PostgreSQL Driver, Version 8.4 to connect to a PostgreSQL 9.2 database.

Step 1. Verify the driver installation.

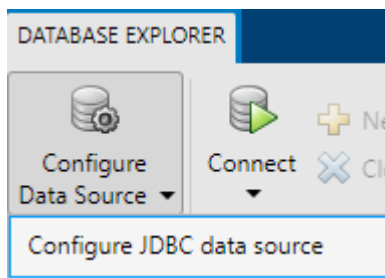
If the JDBC driver for PostgreSQL is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. (This example uses a data source named PostgreSQL.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select PostgreSQL.

JDBC Data Source Configuration

Data Source Details

Name: PostgreSQL

Vendor: Microsoft SQL Server, MySQL, Oracle, PostgreSQL

Driver Location: [Browse]

Connection Parameters

Database: []

Server: localhost

Port Number: 5432

Connection Options

	Name	Value
1		
2		
3		

Buttons: Edit, Test, Save, Delete

Message: []

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Database** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.
- 7 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign **+** to specify additional driver-specific options.
- 8 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 9 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for a PostgreSQL database.

```
vendor = "PostgreSQL";  
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named PostgreSQL, full path of the JDBC driver file /home/user/DB_Drivers/postgresql-8.4-702.jdbc4.jar, database name toystore_doc, database server dbtb00, and port number 5432.

```
opts = setoptions(opts, ...  
    'DataSourceName', "PostgreSQL", ...  
    'JDBCdriverLocation', "/home/user/DB_Drivers/postgresql-8.4-702.jdbc4.jar", ...  
    'DatabaseName', "toystore_doc", 'Server', "dbtb00", ...  
    'PortNumber', 5432);
```

- 3 Test the database connection by specifying the user name username and password pwd, or leave these arguments blank if your database does not require them.

```
username = "username";  
password = "pwd";  
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the PostgreSQL database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to PostgreSQL Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 Select the catalog and schema from the **Catalog** and **Schema** lists. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to PostgreSQL Using JDBC Driver and Command Line

- 1 Connect to a PostgreSQL database using the configured JDBC data source, user name `username`, and password `pwd`.

```
datasource = "PostgreSQL";  
username = "username";  
password = "pwd";  
conn = database(datasource,username,password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `saveAsDataSource` | `setoptions` | `testConnection`

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

PostgreSQL ODBC for Linux

This tutorial shows how to set up a data source and connect to a PostgreSQL database using the command line. The tutorial uses the JDBC4 PostgreSQL Driver, Version 8.4 to connect to a PostgreSQL 9.2 database on the Linux platform.

Step 1. Verify the driver installation.

The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers.

If you need to install an ODBC driver, contact your database administrator or see PostgreSQL Software Catalogue - Drivers and Interfaces.

Step 2. Set up the data source.

Follow the instructions in PostgreSQL Documentation Configuration Files to create a data source name (DSN).

Step 3. Connect using the command line.

- 1 Connect to the database using the configured DSN, user name `username`, and password `pwd` with the `odbc` function. For example, this code assumes that you are connecting to an ODBC data source PostgreSQL.

```
datasource = "PostgreSQL";  
username = "username";  
password = "pwd";  
conn = odbc(datasource,username,password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Functions

`odbc` | `close`

More About

- “Configure Driver and Data Source” on page 2-14
- “PostgreSQL ODBC for Windows” on page 2-56
- “PostgreSQL ODBC for Linux DSN-Less Connection” on page 2-105
- “Modify and Delete Data Sources” on page 4-17
- “Database Connection Error Messages” on page 3-7

PostgreSQL ODBC for Linux DSN-Less Connection

This tutorial shows how to verify your driver installation and connect to a PostgreSQL database using a DSN-less connection string at the command line. (DSN is a data source name.) The tutorial uses the JDBC4 PostgreSQL Driver, Version 8.4 to connect to a PostgreSQL 9.2 database on the Linux platform.

Step 1. Verify the driver installation.

The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers.

Step 2. Connect using the DSN-less connection string and command line.

- 1 Connect to the database using the DSN-less connection string with the `odbc` function. For example, this code assumes that you are connecting to the local database server, database name `toystore_doc`, user name `username`, and password `pwd`.

```
dsnless = strcat("Driver={PostgreSQL ANSI};Servername=localhost;", ...  
                "Database=toystore_doc;UID=username;PWD=pwd");  
conn = odbc(dsnless);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

`odbc` | `close`

Related Examples

- “Configure Driver and Data Source” on page 2-14
- “PostgreSQL ODBC for Linux” on page 2-104
- “PostgreSQL ODBC for Windows” on page 2-56
- “Database Connection Error Messages” on page 3-7

PostgreSQL JDBC for Linux

This tutorial shows how to set up a data source and connect to a PostgreSQL database using the Database Explorer app or the command line. The tutorial uses the JDBC4 PostgreSQL Driver, Version 8.4 to connect to a PostgreSQL 9.2 database.

Step 1. Verify the driver installation.

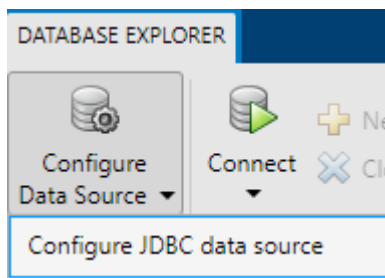
If the JDBC driver for PostgreSQL is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. (This example uses a data source named PostgreSQL.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select PostgreSQL.

JDBC Data Source Configuration

Data Source Details

Name: PostgreSQL

Vendor:

- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL**

Driver Location: ...

Connection Parameters

Database:

Server: localhost

Port Number: 5432

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Database** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.

- 7 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign **+** to specify additional driver-specific options.
- 8 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 9 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for a PostgreSQL database.

```
vendor = "PostgreSQL";  
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named PostgreSQL, full path of the JDBC driver file /home/user/DB_Drivers/postgresql-8.4-702.jdbc4.jar, database name toystore_doc, database server dbtb00, and port number 5432.

```
opts = setoptions(opts, ...  
    'DataSourceName', "PostgreSQL", ...  
    'JDBCDriverLocation', "/home/user/DB_Drivers/postgresql-8.4-702.jdbc4.jar", ...  
    'DatabaseName', "toystore_doc", 'Server', "dbtb00", ...  
    'PortNumber', 5432);
```

- 3 Test the database connection by specifying the user name `username` and password `pwd`, or leave these arguments blank if your database does not require them.

```
username = "username";  
password = "pwd";  
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the PostgreSQL database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to PostgreSQL Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 Select the catalog and schema from the **Catalog** and **Schema** lists. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to PostgreSQL Using JDBC Driver and Command Line

- 1 Connect to a PostgreSQL database using the configured JDBC data source, user name `username`, and password `pwd`.

```
datasource = "PostgreSQL";
username = "username";
password = "pwd";
conn = database(datasource, username, password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `saveAsDataSource` | `setoptions` | `testConnection`

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

SQLite JDBC for macOS

This tutorial shows how to set up a data source and connect to an SQLite database using the Database Explorer app or the command line. The tutorial uses the SQLite JDBC 3.7.2 Driver to connect to an SQLite Version 3.7.17 database.

Step 1. Verify the driver installation.

If the JDBC driver for SQLite is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

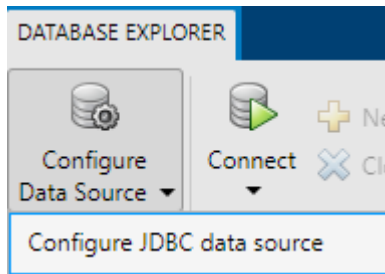
If you do not want to install a driver and want to store relational data quickly, you can use the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. (This example uses a data source named `SQLite`.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select `Other`.

JDBC Data Source Configuration

Data Source Details

Name:

Vendor:

- MySQL
- Oracle
- PostgreSQL
- Other**

Driver Location: ...

Connection Parameters

Driver:

URL:

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Driver** box, enter the SQLite driver Java class object. Here, use `org.sqlite.JDBC`.

Note Your entries for **Driver** and **URL** can vary depending on the type and version of the JDBC driver and your database. For details, see the JDBC driver documentation for your database.

- 7 Connect to the SQLite database by creating a URL string using the format `jdbc:subprotocol:subname`. The `jdbc` part of this string remains constant for any JDBC driver. `subprotocol` is a database type, in this case, `sqlite`. For SQLite, `subname` contains the location of the database. For example, your URL string is `jdbc:sqlite:dbpath`, where `dbpath` is the full path to your SQLite database on your computer. Enter your string in the **URL** box and press **Enter**.

- 8 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign + to specify additional driver-specific options.
- 9 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 10 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for an SQLite database.

```
vendor = "Other";
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named SQLite, full path of the SQLite driver location /home/user/Drivers/sqlite-jdbc-3.8.11.2.jar, SQLite driver Java class object org.sqlite.JDBC, and URL string jdbc:sqlite:/home/user/Databases/sqlite.db.

```
opts = setoptions(opts, ...
    'DataSourceName', "SQLite", ...
    'JDBCDriverLocation', "/home/user/Drivers/sqlite-jdbc-3.8.11.2.jar", ...
    'Driver', "org.sqlite.JDBC", ...
    'URL', "jdbc:sqlite:/home/user/Databases/sqlite.db");
```

- 3 Test the database connection by specifying the user name username and password pwd, or leave these arguments blank if your database does not require them.

```
username = "username";
password = "pwd";
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the SQLite database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to SQLite Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source

name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 3 Select tables in the **Data Browser** pane to query the database.
- 4 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to SQLite Using JDBC Driver and Command Line

- 1 Connect to an SQLite database using the configured JDBC data source, user name `username`, and password `pwd`.

```
datasource = "SQLite";
username = "username";
password = "pwd";
conn = database(datasource, username, password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `saveAsDataSource` | `setoptions` | `testConnection`

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

SQLite JDBC for Linux

This tutorial shows how to set up a data source and connect to an SQLite database using the Database Explorer app or the command line. The tutorial uses the SQLite JDBC 3.7.2 Driver to connect to an SQLite Version 3.7.17 database.

Step 1. Verify the driver installation.

If the JDBC driver for SQLite is not installed on your computer, find the link on the Driver Installation page to install the driver. Follow the instructions to download and install this driver on your computer.

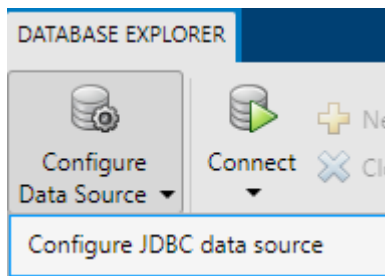
If you do not want to install a driver and want to store relational data quickly, you can use the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Step 2. Set up the data source.

You set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a name for your data source. (This example uses a data source named `SQLite`.) You use this name to establish a connection to your database.
- 4 From the **Vendor** list, select `Other`.

JDBC Data Source Configuration

Data Source Details

Name:

Vendor:
 MySQL
 Oracle
 PostgreSQL
 Other

Driver Location: ...

Connection Parameters

Driver:

URL:

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

- 5 In the **Driver Location** box, enter the full path to the JDBC driver file.
- 6 In the **Driver** box, enter the SQLite driver Java class object. Here, use `org.sqlite.JDBC`.

Note Your entries for **Driver** and **URL** can vary depending on the type and version of the JDBC driver and your database. For details, see the JDBC driver documentation for your database.

- 7 Connect to the SQLite database by creating a URL string using the format `jdbc:subprotocol:subname`. The `jdbc` part of this string remains constant for any JDBC driver. `subprotocol` is a database type, in this case, `sqlite`. For SQLite, `subname` contains the location of the database. For example, your URL string is `jdbc:sqlite:dbpath`, where `dbpath` is the full path to your SQLite database on your computer. Enter your string in the **URL** box and press **Enter**.
- 8 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign **+** to specify additional driver-specific options.
- 9 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database, or leave these boxes blank if your database does not require them. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 10 Click **Save**. The JDBC Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a JDBC data source for an SQLite database.

```
vendor = "Other";
opts = databaseConnectionOptions("jdbc", vendor);
```

- 2 Set the JDBC connection options. For example, this code assumes that you are connecting to a JDBC data source named `SQLite`, full path of the SQLite driver location `/home/user/Drivers/sqlite-jdbc-3.8.11.2.jar`, SQLite driver Java class object `org.sqlite.JDBC`, and URL string `jdbc:sqlite:/home/user/Databases/sqlite.db`.

```
opts = setoptions(opts, ...
    'DataSourceName', "SQLite", ...
    'JDBCDriverLocation', "/home/user/Drivers/sqlite-jdbc-3.8.11.2.jar", ...
    'Driver', "org.sqlite.JDBC", ...
    'URL', "jdbc:sqlite:/home/user/Databases/sqlite.db");
```

- 3 Test the database connection by specifying the user name `username` and password `pwd`, or leave these arguments blank if your database does not require them.

```
username = "username";
password = "pwd";
status = testConnection(opts, username, password);
```

- 4 Save the JDBC data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the SQLite database using the Database Explorer app or the JDBC driver and command line.

Step 3. Connect using the Database Explorer app or the command line.

Connect to SQLite Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.

- 2 In the connection dialog box, enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 3 Select tables in the **Data Browser** pane to query the database.
- 4 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to SQLite Using JDBC Driver and Command Line

- 1 Connect to an SQLite database using the configured JDBC data source, user name `username`, and password `pwd`.

```
datasource = "SQLite";
username = "username";
password = "pwd";
conn = database(datasource,username,password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `saveAsDataSource` | `setoptions` | `testConnection`

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

Other ODBC-Compliant or JDBC-Compliant Databases

This tutorial provides steps for configuring data sources and connecting to other ODBC-compliant or JDBC-compliant databases that are not listed in “Configure Driver and Data Source” on page 2-14. After creating a data source, you can connect to your database using the Database Explorer app or the command line.

ODBC-Compliant Databases

These steps show how to configure a driver and connect to an ODBC-compliant database. Database Toolbox can connect to any ODBC-compliant database that is relational and uses ANSI® SQL. For example, if your database is Microsoft Excel® or IBM DB2®, follow these basic steps:

- 1** If your driver is not preinstalled on your computer, find a compatible driver and install it on your computer. You can view preinstalled drivers using the Microsoft ODBC Data Source Administrator dialog box. For details about this dialog box, see Driver Installation.
- 2** Create a data source by using the installed driver and the Microsoft ODBC Data Source Administrator dialog box. To open this dialog box, use the `configureODBCDataSource` function.
- 3** Use the Database Explorer app to test your connection and connect to your database. For an example, see “MySQL ODBC for Windows” on page 2-47.

Or, you can connect to your database using the `database` function at the command line.

- 4** For detailed assistance, contact your database administrator or refer to your database documentation.

JDBC-Compliant Databases

You set up a data source to a JDBC-compliant database using the Database Explorer app or the command line.

These steps show how to configure a driver and connect to a JDBC-compliant database. Database Toolbox can connect to any JDBC-compliant database that is relational and uses ANSI SQL. For example, if your database is Apache Derby or Microsoft Windows Azure, follow these basic steps:

Note The details of these steps can vary depending on your database and database version. For detailed assistance, contact your database administrator or refer to your database documentation.

Set Up Data Source Using Database Explorer

- 1** If your driver is not installed on your computer, find a compatible driver and install it on your computer.
- 2** Create a data source by using the JDBC Data Source Configuration dialog box. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure JDBC data source** to open the dialog box.
- 3** In the **Name** box, enter a name for your data source. You use this name to establish a connection to your database.
- 4** From the **Vendor** list, select **Other**.

- 5 In the **Driver Location** box, specify the full path of the driver file.
- 6 To connect to a JDBC-compliant database, you must know your database driver Java class object. For example, the Java class object for an SQLite database driver is `org.sqlite.JDBC`. In the **Driver** box, specify the driver value.
- 7 To connect to a JDBC-compliant database, you must create a URL string. The URL string has the form `jdbc:subprotocol:subname`. The `jdbc` part of this string remains constant for any JDBC driver. `subprotocol` is the database type. The `subname` contains the location of the database and additional connection information, such as the port number. For example, if you are using SQLite, the URL string is `jdbc:sqlite:dbpath`, where `dbpath` is the full path to your SQLite database on your computer. In the **URL** box, specify the URL string.

Note For JDBC-compliant databases, specify the driver and URL in the JDBC Data Source Configuration dialog box. The driver and the URL string can vary depending on the type and version of the JDBC driver and your database. For details about the driver and URL, see the JDBC driver documentation for your database.

- 8 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign **+** to specify additional driver-specific options.
- 9 Click **Test** to test your connection. If the connection is successful, then click **Save** to save the JDBC data source.
- 10 Connect to your database with the saved JDBC data source by using the Database Explorer app. For an example, see “SQLite JDBC for Windows” on page 2-65.

Set Up Data Source Using Command Line

- 1 If your driver is not installed on your computer, find a compatible driver and install it on your computer.
- 2 Create a data source using the `databaseConnectionOptions` function.
- 3 Set the JDBC connection options using the `setoptions` function.
- 4 Test the database connection using the `testConnection` function.
- 5 Save the JDBC data source using the `saveAsDataSource` function.
- 6 Connect to your database with the saved JDBC data source by using the `database` function. For an example, see “SQLite JDBC for Windows” on page 2-65.

See Also

Apps

Database Explorer

Functions

`database` | `close` | `databaseConnectionOptions` | `configureODBCDataSource` | `setoptions` | `testConnection` | `saveAsDataSource`

Related Examples

- “MySQL ODBC for Windows” on page 2-47
- “SQLite JDBC for Windows” on page 2-65

More About

- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Java Class Path”
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13
- “Database Connection Error Messages” on page 3-7

Connect to Database

To connect to a database from MATLAB, install an ODBC or JDBC driver and create a data source. Or, you can connect to MySQL or PostgreSQL databases using the native interfaces. For details about driver installation and data source setup, see “Configure Driver and Data Source” on page 2-14. If you do not have an installed database and want to store relational data quickly, you can use the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

You can connect to a database using the **Database Explorer** app or the command line. These two options enable you to perform different actions. For details about deciding which option to use, see “Connection Options” on page 2-8.

Database Explorer App Connection Workflow

Use these steps as a general workflow for creating a database connection using the app.

- In the **Data Source** section of the **Database Explorer** tab, click **Configure Data Source** and select the appropriate option for configuring an ODBC, JDBC, or native data source.
- In the **Connections** section, click **Connect** and select the configured data source to create a database connection.
- To close the database connection, in the **Connections** section, click **Close Connection**. If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Command Line Connection Workflow

Use these steps as a general workflow for creating a database connection at the command line.

- Configure an ODBC data source using the `configureODBCDataSource` function, or configure a JDBC data source using the `databaseConnectionOptions` function.
- Create a database connection by completing one of these steps:
 - Create an ODBC database connection using the `odbc` function.
 - Create an ODBC or JDBC database connection using the `database` function.
 - Create a MySQL database connection using the MySQL native interface with the `mysql` function.
 - Create a PostgreSQL database connection using the PostgreSQL native interface with the `postgresql` function.
- If you do not configure a data source using a data source name (DSN), you can create a DSN-less connection using a connection string with the `odbc` function.
- Close the database connection using the `close` function.

Database List

After choosing to use the Database Explorer app or the command line, connect to your database by following the steps for the corresponding database and driver type, as given in this table.

Database	Driver or Interface	Connection Topics
Microsoft Access	ODBC	<p>“Connect to Access Using Database Explorer App” on page 2-21</p> <p>“Connect to Access Using ODBC Driver and Command Line” on page 2-22</p>
Microsoft SQL Server	ODBC	<p>“Connect to SQL Server Using Database Explorer App” on page 2-26</p> <p>“Connect to SQL Server Using ODBC Driver and Command Line” on page 2-27</p> <p>Connect to SQL Server Using DSN-Less Connection String and Command Line on page 2-28</p>
	JDBC	<p>“Connect to SQL Server Using Database Explorer App” on page 2-34</p> <p>“Connect to SQL Server Using JDBC Driver and Command Line” on page 2-34</p>
Oracle	ODBC	<p>“Connect to Oracle Using Database Explorer App” on page 2-39</p> <p>“Connect to Oracle Using ODBC Driver and Command Line” on page 2-39</p>
	JDBC	<p>“Connect to Oracle Using Database Explorer App” on page 2-45</p> <p>“Connect to Oracle Using JDBC Driver and Command Line” on page 2-45</p>

Database	Driver or Interface	Connection Topics
MySQL	ODBC	<p>“Connect to MySQL Using Database Explorer App” on page 2-49</p> <p>“Connect to MySQL Using ODBC Driver and Command Line” on page 2-50</p> <p>Connect to MySQL Using DSN-Less Connection String and Command Line on page 2-51</p>
	MySQL native interface	<p>“Connect to MySQL Using Database Explorer App” on page 6-4</p> <p>“Connect to MySQL Using Command Line” on page 6-4</p>
	JDBC	<p>“Connect to MySQL Using Database Explorer App” on page 2-54</p> <p>“Connect to MySQL Using JDBC Driver and Command Line” on page 2-55</p>
PostgreSQL	ODBC	<p>“Connect to PostgreSQL Using Database Explorer App” on page 2-58</p> <p>“Connect to PostgreSQL Using ODBC Driver and Command Line” on page 2-58</p> <p>Connect to PostgreSQL Using DSN-Less Connection String and Command Line on page 2-60</p>
	PostgreSQL native interface	<p>“Connect to PostgreSQL Using Database Explorer App” on page 7-4</p> <p>“Connect to PostgreSQL Using Command Line” on page 7-5</p>
	JDBC	<p>“Connect to PostgreSQL Using Database Explorer App” on page 2-63</p> <p>“Connect to PostgreSQL Using JDBC Driver and Command Line” on page 2-64</p>

Database	Driver or Interface	Connection Topics
SQLite	JDBC	<p>“Connect to SQLite Using Database Explorer App” on page 2-68</p> <p>“Connect to SQLite Using JDBC Driver and Command Line” on page 2-68</p>
	None	To create an SQLite connection when no driver or database installation is required, use the <code>sqlite</code> function to create a new SQLite database file or connect to an existing SQLite database file.
All other ODBC-compliant or JDBC-compliant databases	ODBC or JDBC	“Other ODBC-Compliant or JDBC-Compliant Databases” on page 2-118

See Also

`database` | `close` | `configureODBCDataSource` | `databaseConnectionOptions` | `odbc`

More About

- “Setup Requirements for Database Connection” on page 2-11
- “Access Relational Database Data in MATLAB” on page 2-2
- “Connection Options” on page 2-8
- “Choose Between ODBC and JDBC Drivers” on page 2-12
- “Configure Driver and Data Source” on page 2-14
- “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5

Data Import Using Database Explorer App or Command Line

You can import data from a database into MATLAB using the **Database Explorer** app or the command line. To select data for import, you can build an SQL query visually by using the Database Explorer app. Or, you can use the command line to write SQL queries. To achieve maximum performance with large data sets, use the command line instead of the Database Explorer app.

After importing data, you can repeat the steps in the process, such as connecting to a database, executing an SQL query, and so on, by using a MATLAB script to automate them.

To open multiple connections to the same database simultaneously, you can create multiple SQL queries using the Database Explorer app. Or, you can connect to the database using the command line.

If you do not have access to a database and want to import your data quickly, you can use the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Data Import Using Database Explorer App

If you have minimal proficiency writing SQL queries or want to browse the data in a database quickly, use the **Database Explorer** app. To build queries, see “Create SQL Queries Using Database Explorer App” on page 4-2. After creating a query using the Database Explorer app, you can generate the SQL code for the query. For details, see “Generate SQL Query” on page 4-20. You can embed the generated SQL code into the SQL query that you specify in the `fetch` function. Or, you can create an SQL script file to use with the `executeSQLScript` function.

If you want to automate the current task after you create the SQL query, then generate a MATLAB script. For details, see “Generate MATLAB Script” on page 4-20.

Data Import Using Command Line

If you are not familiar with writing SQL queries, then use the **Database Explorer** app to select data to import from your database. Or, you can use the `sqlread` function at the command line. This function needs only a database connection and the database table name to import data. Furthermore, the `sqlread` function does not require you to set database preferences.

If you know how to write SQL queries, you can write basic SQL statements as character vectors or string scalars. For a simple example, see “Import Data from Database Table Using `sqlread` Function” on page 5-58.

When writing SQL queries, you can import data into MATLAB in one of two ways. Use the `select` function for maximum memory efficiency and quick access to imported data. Or, use the `fetch` function to import numeric data with double precision by default or define the import strategy for the SQL query.

For memory management, see “Data Import Memory Management” on page 5-20.

If you have a stored procedure that imports data, then use the `runstoredprocedure` or `fetch` functions.

Custom Data Types

When importing data from a database, Database Toolbox functions return custom data types, such as Oracle ref cursors, as Java objects. You can manually parse these objects to retrieve their data contents. Use the `methods` function to access all the methods of a Java object. Use the available methods to retrieve data from a Java object. The steps for your object are specific to your database. For details, refer to your JDBC driver or database documentation.

SQL Queries Saved in Scripts or Files

If you have a long SQL query or multiple SQL queries that you want to run sequentially to import data, create an SQL script file containing your SQL queries. To execute the SQL script file, use the `executeSQLScript` function. If you have SQL queries stored in `.sql` or text files that you want to run from MATLAB, you can also use this function.

See Also

`select` | `fetch` | `database` | `sqlread`

More About

- “Connection Options” on page 2-8
- “Data Import Memory Management” on page 5-20
- “Working with Large Data Sets” on page 2-127
- “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5
- “Data Type Support” on page 1-3
- “Data Retrieval Restrictions” on page 1-5

Working with Large Data Sets

Connect to a Database with Maximum Performance

When you are using MATLAB with a database containing large volumes of data, you can experience out-of-memory issues or slow processing. To achieve the fastest performance, connect to your database using the native ODBC interface. If the native ODBC interface does not work, connect to your database using a JDBC driver. For details, see “Connect to Database” on page 2-121.

Import Large Data Sets into MATLAB

If you are selecting large volumes of data in a database to import into MATLAB, you can experience out-of-memory issues or slow processing. To achieve the fastest performance, you can import the data in batches.

When working with a native ODBC connection, the amount of memory available to MATLAB can restrict you from processing your whole set of data at once. To manage the MATLAB memory, process your data in parts. Use the `fetch` function to limit the number of rows your query returns by using the 'MaxRows' input argument. Using a MATLAB script, you can import data in increments until all data is retrieved. For an example, see `fetch`.

If you do not have access to a database and want to import large data sets, you can use the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Export Large Data Sets from MATLAB

When inserting large volumes of data into a database, you can experience slow processing. To achieve the fastest performance, use the `sqlwrite` function to export your data from MATLAB.

If you do not have access to a database and want to export large data sets, you can use the `insert` function with the MATLAB interface to SQLite. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Access Large Data Using a DatabaseDatastore

An alternative for importing large data sets stored in a database into MATLAB is using a `DatabaseDatastore`. A `DatabaseDatastore` is a datastore that contains a collection of data stored in a database.

You can analyze data in a `DatabaseDatastore` using tall arrays with common MATLAB functions, such as `mean` and `histogram`. For details, see “Analyze Large Data in Database Using Tall Arrays” on page 5-30. Or, for more control, you can also write your own algorithms using `MapReduce`. For details, see “Analyze Large Data in Database Using MapReduce” on page 5-27.

See Also

More About

- “Choose Between ODBC and JDBC Drivers” on page 2-12

- “Data Import Using Database Explorer App or Command Line” on page 2-125
- “Import Data from Database Table Using sqlread Function” on page 5-58
- “Insert Data into Database Table” on page 5-61

Working with Data Sources

- “Writing Data Common Errors” on page 3-2
- “Importing Data Common Errors” on page 3-3
- “Database Connection Error Messages” on page 3-7
- “Database Explorer App Error Messages” on page 3-13
- “SQL Prepared Statement Error Messages” on page 3-15

Writing Data Common Errors

This table describes how to address common errors you might encounter while working with the `sqlwrite` function. These errors apply to all database vendors.

Error Message	Probable Causes	Resolution
<code>columnname</code> column value must be a numeric array or cell array of numeric scalars.	The specified data type of the database column is invalid.	Specify a valid data type for the database column. For valid data types, see the <code>data</code> input argument description in the <code>sqlwrite</code> function.
<code>columnname</code> column value must be a datetime array, cell array of character vectors, or string array.		
<code>columnname</code> column value must be a logical array.		
<code>columnname</code> column value must be a cell array of character vectors or string array.		
JDBC/ODBC Error: <code>errormessage</code>	The JDBC or ODBC driver throws an error.	Consult your database driver documentation.
Unable to create <code>tablename</code> without column types. Specify 'ColumnType' for each variable in the table data.	You are creating an empty database table.	Specify the 'ColumnType' name-value pair argument and provide the data type for all columns in the database table. For details, see the <code>sqlwrite</code> function.
Specify 'ColumnType' for each variable in the table data.	You did not specify the data type of at least one column in the database table.	Specify the 'ColumnType' name-value pair argument and provide the data type for all columns in the database table. For details, see the <code>sqlwrite</code> function.

Importing Data Common Errors

Address common errors that you can encounter when importing data from databases and customizing import options.

Data Import Common Errors

The following table describes errors that can occur in either the Database Explorer app or the command line when you use the `fetch`, `sqlinnerjoin`, and `sqlouterjoin` functions.

Vendor	Error Message	Probable Causes	Resolution
All	Must provide either the "Keys" value, or both the "LeftKeys" and "RightKeys" values.	You specified only the 'LeftKeys' or 'RightKeys' name-value pair argument.	Specify the 'Keys' name-value pair argument, or both the 'LeftKeys' and 'RightKeys' name-value pair arguments.
	Multiple table entry found for <i>tablename</i> . Must provide <i>LeftCatalog/RightCatalog</i> and <i>LeftSchema/RightSchema</i> values.	The database contains multiple tables with the same name across catalogs and schemas.	Specify the 'LeftCatalog' and 'LeftSchema' or 'RightCatalog' and 'RightSchema' name-value pair arguments.
	Unable to find information for table <i>tablename</i> . Must provide either the "Keys" value, or both the "LeftKeys" and "RightKeys" values.	The function cannot find information about the specified database table.	Specify the 'Keys' name-value pair argument, or both the 'LeftKeys' and 'RightKeys' name-value pair arguments.
	Unable to find columns for table <i>tablename</i> . Must provide either the "Keys" value, or both the "LeftKeys" and "RightKeys" values.	The function cannot find information about the columns of the specified database table.	Specify the 'Keys' name-value pair argument, or both the 'LeftKeys' and 'RightKeys' name-value pair arguments.

Vendor	Error Message	Probable Causes	Resolution
	Unable to find common keys for table <i>lefttable</i> and <i>righttable</i> . Must provide either the "Keys" value, or both the "LeftKeys" and "RightKeys" values.	The function cannot find common keys between the specified left and right tables to join.	Specify the 'Keys' name-value pair argument, or both the 'LeftKeys' and 'RightKeys' name-value pair arguments.
	The number of key variables on the left and right must be the same.	The number of specified keys for the left and right tables do not match.	Specify the same number of keys for the 'LeftKeys' and 'RightKeys' name-value pair arguments.
Microsoft SQL Server	The statement did not return a result set.	There are other SQL statements in the middle of the stored procedure. This error happens after you execute <code>exec</code> but before you execute <code>fetch</code> . This error happens only with the command line.	Add 'SET NOCOUNT ON' at the beginning of your stored procedure. For details, see <code>exec</code> .
Microsoft SQL Server	JDBC Driver 3.0 returns incorrect date values when used with JRE™ 1.7 by a Java application.	There is an issue with the Microsoft SQL Server JDBC Driver 3.0. This error happens after you execute <code>fetch</code> . This error happens either with Database Explorer or the command line.	Install a hotfix from Microsoft for JDBC Driver 3.0. Alternatively, upgrade your Microsoft SQL Server JDBC driver to version 4.0.
Microsoft SQL Server	Connection is busy with results for another command.	You are connecting to Microsoft SQL Server using a driver that <code>preview</code> does not support.	Connect to Microsoft SQL Server using the JDBC driver.
Oracle	Stored procedures and functions return result sets as cursor types.	The JDBC driver returns stored procedure and function result sets as custom Java objects. This error happens after you execute <code>fetch</code> . This error happens only with the command line.	Write custom MATLAB code to process the Java objects into MATLAB variables.

Vendor	Error Message	Probable Causes	Resolution
PostgreSQL	Java exception occurred: java.lang.OutOfMemoryError: Java heap space	The JDBC driver caches results in the memory. There is not enough memory in the Java heap to store the large amount of data fetched from your database. This error happens after you execute fetch. This error happens either with Database Explorer or the command line.	Write custom code. Write the code for connecting to your database via the command line. Then write the following. <pre>conn.setAutoCommit = 'off'; h = conn.Handle; stmt = h.createStatement(); stmt.setFetchSize(50); rs = stmt.executeQuery(java.lang.String(' SELECT * FROM largeData where productnumber <= 3000000'));</pre> Modify the previous statement to include your SQL query instead. Then process the result set object rs in batches.

Custom Import Options Common Errors

The following table describes errors that can occur when you use the `SQLImportOptions` object to customize options for importing data from a database. These error messages apply across all database vendors.

Error Message	Probable Causes	Resolution
Calling <i>function</i> without an output argument has no effect. Use the following instead: <code>opts = function(opts,...)</code>	You did not specify an output argument when executing the <code>setoptions</code> function.	Use the <code>setoptions</code> function with an output argument.
<i>argument</i> must be a character vector or cell array of character vectors.	The specified input argument has an invalid data type.	The input argument must be a character vector or cell array of character vectors.

Error Message	Probable Causes	Resolution
Unknown variable name: <code>'argument'</code> .	The specified variable name is invalid.	Specify a variable name that exists in the <code>VariableNames</code> property of the <code>SQLImportOptions</code> object.
Variable selection out of range. Vector must contain integers between 1 and N, where N is the number of variables in the import options.	The specified index value is out of bounds within the number of selected variables.	Specify an index that is in the range of the number of variables in the <code>SelectedVariableNames</code> property of the <code>SQLImportOptions</code> object.
Expected a name or numeric index of a variable name.	The data type of the specified input argument is invalid.	The input argument must be a numeric index or a variable name.
Cell array of types must be a vector of length <i>n</i> .	The length of the specified data types is invalid.	When you set the <code>VariableTypes</code> property of the <code>SQLImportOptions</code> object, the length of the cell array must be equal to the number of variables.
Cell array of names must be a vector of length <i>n</i> .	The length of the specified variable names is invalid.	When you set the <code>VariableNames</code> property of the <code>SQLImportOptions</code> object, the length of the cell array must be equal to the number of variables.

See Also

`fetch` | `setoptions` | `getoptions` | `sqlinnerjoin` | `sqlouterjoin`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Data Import Using Database Explorer App or Command Line” on page 2-125
- “Call Stored Procedure That Returns Data” on page 5-17

Database Connection Error Messages

This table describes how to address common errors you might encounter while connecting to the Database Toolbox using Database Explorer or the command line.

Connection Error Messages and Probable Causes

Vendor	Error Message	Probable Causes	Resolution
All	Undefined variable 'database' or class 'database.ODBCConnection'.	<ul style="list-style-type: none"> Database Toolbox software is not installed. You are connecting using the native ODBC interface with MATLAB R2013a or earlier. 	<ul style="list-style-type: none"> Ensure that Database Toolbox software is installed. If you want to use the native ODBC interface, ensure that MATLAB R2013b or later is installed.
	Unable to access data source name. Use databaseConnectionOptions to create a JDBC data source. Use configureODBCDataSource to create an ODBC data source.	<p>The specified data source name does not exist.</p> <p>This error message occurs with Windows only.</p>	Create a JDBC or ODBC data source using the corresponding databaseConnectionOptions or configureODBCDataSource function. For examples, see “Configure Driver and Data Source” on page 2-14.
	Parameter name must be 'AutoCommit', 'ReadOnly', 'LoginTimeout', 'ErrorHandling'.	The specified name-value pair arguments are invalid with the database(datasource, username, password, Name, Value) syntax.	Specify one or more of these valid name-value pair arguments with the database function: 'AutoCommit', 'ReadOnly', 'LoginTimeout', and ErrorHandling.
All ODBC-Compliant Databases	[Microsoft][ODBC Driver Manager] Data source name not found and no default driver specified	Data source name is not spelled correctly.	Verify your data source name.
	[Microsoft][ODBC Driver Manager] The specified DSN contains an architecture mismatch between Driver and Application	There is a difference in the bitness (32-bit or 64-bit) between the database, driver, and MATLAB.	<p>Use a 64-bit driver. If you have issues working with the ODBC driver, use the JDBC driver instead. For details about driver installation, see “Configure Driver and Data Source” on page 2-14.</p> <p>To address differences in bitness for Microsoft Access, see “Microsoft Access ODBC for Windows” on page 2-19.</p>

Vendor	Error Message	Probable Causes	Resolution
All JDBC-Compliant Databases	Unable to find JDBC driver file on MATLAB Java class path.	You specify a path to the JDBC driver JAR file that is not on the static or dynamic class path. Or, you specify an incorrect driver name in the Driver box of the JDBC Data Source Configuration dialog box.	Specify the full path to the JDBC driver file in the Driver Location box and the correct driver name in the Driver box of the JDBC Data Source Configuration dialog box. For details, see the <code>databaseConnectionOptions</code> function.
	Unable to access data source name. Use <code>databaseConnectionOptions</code> to create a JDBC data source.	The specified data source name does not exist. This error message occurs with UNIX® only.	Create a JDBC data source using the <code>databaseConnectionOptions</code> function. For examples, see “Configure Driver and Data Source” on page 2-14.
	JDBC data source does not contain driver location. Use <code>databaseConnectionOptions</code> to specify JDBC driver location.	The specified driver location in the JDBC data source is invalid.	Modify the JDBC data source to specify a valid JDBC driver location using the <code>databaseConnectionOptions</code> function.
Microsoft Access	[Microsoft][ODBC Microsoft Access Driver] '(unknown)' is not a valid path. make sure that the path name is spelled correctly and that you are connected to the server on which the file resides	Error occurs in the Connection Failure dialog box after clicking Connect in the Connect to a Data Source dialog box. The file location of the Microsoft Access database is incorrect.	Verify the location of the database file. If the database file is on a network drive, map to the network drive. Modify the existing file location by selecting New > ODBC and selecting the existing database name from the ODBC Data Source Administrator dialog box. Then select Configure to change the database file location.

Vendor	Error Message	Probable Causes	Resolution
Microsoft SQL Server	The TCP/IP connection to the host <i>hostname</i> , port <i>portnumber</i> has failed. Error: "null. Verify the connection properties, check that an instance of SQL Server is running on the host and accepting TCP/IP connections at the port, and that no firewall is blocking TCP connections to the port."	Incorrect server name or port number.	Verify your database server name and your port number. Microsoft SQL Server uses a dynamic port for JDBC. Verify the value using Microsoft SQL Server Configuration Manager. For details, see "Step 2. Verify the port number." on page 2-29
Microsoft SQL Server	This driver is not configured for integrated authentication.	The Microsoft SQL Server Windows authentication library is not added to <code>javainstrumentpath.txt</code> .	Add the Microsoft SQL Server Windows authentication library to <code>javainstrumentpath.txt</code> . For details about configuring a Microsoft SQL Server Authenticated Database Connection, see "Microsoft SQL Server JDBC for Windows" on page 2-29.
Microsoft SQL Server	Invalid string or buffer length.	64-bit ODBC driver error.	Use a JDBC driver or the native ODBC interface instead.
Microsoft SQL Server	JDBC Driver Error: <code>com.microsoft.sqlserver.jdbc.SQLServerDriver.Driver Not Found/Loaded.</code>	The full path to the JAR file was not added to the <code>java.classpath.txt</code> file, or it was added using the <code>java.addpath</code> command. Alternatively, the path to the JAR file is incorrect.	Ensure that the path to the JAR file is not misspelled. Ensure that you add the path to the static class path.
Microsoft SQL Server	<code>com.microsoft.sqlserver.jdbc.AuthenticationJNI <clinit> WARNING: Failed to load the sqljdbc_auth.dll</code>	The path to the folder containing the file <code>sqljdbc_auth.dll</code> was not added to the <code>javainstrumentpath.txt</code> file. Or, the full path to the file was added instead of the path to the folder. This error also occurs when you add the path to the 32-bit version of the DLL using a 64-bit version of MATLAB.	Add the path to the folder containing the file <code>sqljdbc_auth.dll</code> to the <code>javainstrumentpath.txt</code> file. For details about configuring a Microsoft SQL Server Authenticated Database Connection, see "Microsoft SQL Server JDBC for Windows" on page 2-29.

Vendor	Error Message	Probable Causes	Resolution
Microsoft SQL Server	Login failed for user 'DOMAIN\username'.	Either the login credentials you are using are incorrect or your user account does not have enough rights to access the remote machine. This error also occurs when the database server is not configured to accept Integrated Windows Authentication login credentials.	Ensure that your user name and password are correct. Refer to your system administrator for appropriate access rights to your machines. Contact your database administrator to see if your database is set up with Windows Authentication.
Microsoft SQL Server	MSSQLSERVER_ <i>number</i>	The Microsoft SQL Server driver returns a numbered error message.	Find more information about the specific error in System Error Messages.
MySQL	Access denied for user 'user'@'machinename' (using password: YES)	Incorrect user name and password combination.	Verify your user name and password.
MySQL	Communications link failure. The last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets from the server.	Incorrect server name or port number.	Verify your database server name and port number.
MySQL	Unknown database 'databasename'.	Provided database name is incorrect.	Verify your database name.
MySQL	ERROR <i>number</i> (SQLSTATE): <i>errormessage</i>	The MySQL driver returns an error that contains an error number, a SQLSTATE value, and an error message.	Navigate to the latest database documentation in the MySQL Documentation, and search for the specific error.

Vendor	Error Message	Probable Causes	Resolution
Oracle	Error when connecting to Oracle oci8 database using JDBC driver: Error using com.mathworks.toolbox.database.databaseConnect/makeDatabaseConnection Java exception occurred: java.lang.UnsatisfiedLinkError: no ocijdbc11 in java.library.path at java.lang.ClassLoader.loadLibrary(Unknown Source) at java.lang.Runtime.loadLibrary0.....	MATLAB cannot find the Oracle DLL that the oci8 drivers need.	Add the path for the location of the Oracle DLLs to <code>java.library.path.txt</code> . For details, see “Oracle JDBC for Windows” on page 2-41.
Oracle	Invalid Oracle URL specified: OracleDataSource.makeURL	The <code>DriverType</code> parameter is not specified.	Specify the <code>DriverType</code> parameter as either <code>thin</code> for connecting without Windows authentication or <code>oci</code> for connecting with Windows authentication.
Oracle	The Network Adapter could not establish the connection.	Either <code>Server</code> or <code>Portnumber</code> is not specified or has an incorrect value.	Verify the server name and port number for your Oracle database.
Oracle	TNS:listener does not currently know of SID given in connect descriptor: Incorrect database name or incorrect URL.	The service name for your database is incorrect.	Verify the service name for your Oracle database.
Oracle	<i>ORA-number</i>	The Oracle driver returns a numbered error message.	Navigate to the latest database documentation in the Oracle Documentation, and search for the specific error.

See Also
database

More About

- “Configure Driver and Data Source” on page 2-14
- “Connect to Database” on page 2-121

Database Explorer App Error Messages

This table describes how to address common errors you can encounter while working with the **Database Explorer** app. For Database Toolbox connection errors, see “Database Connection Error Messages” on page 3-7.

Error Category	Error Message	Probable Causes	Resolution
Database connection	No data sources found. Configure a data source before creating a new query.	You clicked New Query before configuring any data sources.	Configure at least one ODBC or JDBC data source. For details, see “Configure Driver and Data Source” on page 2-14.
Configure JDBC data source	Unable to find the JDBC driver file on the MATLAB Java class path.	The JDBC Data Source Configuration dialog box displays this message at the bottom of the dialog box. This message is displayed when the JDBC driver for the database, or the driver specified for the Driver option when the Vendor option is set to OTHER, is not found on the MATLAB Java class path.	Specify the full path to the JDBC driver file in the Driver Location box of the JDBC Data Source Configuration dialog box. For details, see the <code>databaseConnectionOptions</code> function.
SQL statements	Database Explorer supports one SELECT SQL statement only.	You entered an incompatible SQL query. The Database Explorer app accepts a single SELECT statement only. Other SQL statements are not supported.	Create one valid SQL SELECT statement in the SQL Query pane.
Data preview	Received the following message from the database:	If an error occurs on the database server when it executes the SQL query, an error message appears in the Data Preview pane. This text precedes the exact error message from the database server.	Refer to the error message to determine the issue. Consult your database administrator for further details.
Data preview	Error occurred while executing the query on the database.	The Database Explorer app returned a MATLAB error while trying to execute the SQL query.	Click the button to view the stack trace. For questions, contact technical support.

Error Category	Error Message	Probable Causes	Resolution
Data preview	No data was returned for this SQL Query.	<p>The SQL query executed successfully but did not return any data.</p> <p>The Database Explorer app displays this error message in the Data Preview pane when previewing data or in a dialog box when importing data.</p>	<p>Use the Database Explorer app to modify the SQL query.</p> <p>Ensure that the selected table contains data by clicking it in the Data Browser pane.</p>
Data import	<i>variable</i> not a valid MATLAB variable name.	You entered an invalid MATLAB variable name.	Enter a valid MATLAB variable name, such as <code>data</code> .

See Also

Apps

Database Explorer

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14

SQL Prepared Statement Error Messages

This table describes how to address common errors you can encounter when working with SQL prepared statements. For details about SQL prepared statements, see the `SQLPreparedStatement` object.

Error Message	Probable Causes	Resolution
Invalid SQL statement. SQL prepared statement must be SELECT, INSERT, UPDATE, DELETE, or CALL.	The SQL prepared statement is invalid.	Specify a valid SQL prepared statement in the <code>SQLPreparedStatement</code> object by using the <code>databasePreparedStatement</code> function. You can specify these SQL statements: <ul style="list-style-type: none"> • SELECT • INSERT • UPDATE • DELETE • CALL
Invalid SQL statement. SQL prepared statement must have at least one SQL parameter.	The specified SQL prepared statement is invalid because it has no parameters.	Specify at least one parameter value to bind in the SQL prepared statement by using the <code>databasePreparedStatement</code> function.
Parameter selection out of range. Vector must contain integers between 1 and n, where n is the number of parameters.	The specified indices for the selected parameters in the <code>bindParamValues</code> function are invalid.	Specify indices for the selected parameters within the range of the parameter values in the <code>bindParamValues</code> function. The values must be between 1 and n, where n is the number of parameters.
Parameter index must be a numeric scalar.	The specified indices for the selected parameters in the <code>bindParamValues</code> function are not numeric.	Specify numeric index values for the selected parameters in the <code>bindParamValues</code> function. The values must be between 1 and n, where n is the number of parameters.
Value must be a cell array with length between 1 and n, where n is the number of parameters.	The parameter values in the <code>bindParamValues</code> function are invalid.	The values to bind must be a cell array that has a length between 1 and n, where n is the number of parameters.

See Also

`databasePreparedStatement` | `bindParamValues` | `close`

More About

- “Import Data Using SQL Prepared Statement with Multiple Parameter Values” on page 5-75

Using Database Explorer

Create SQL Queries Using Database Explorer App

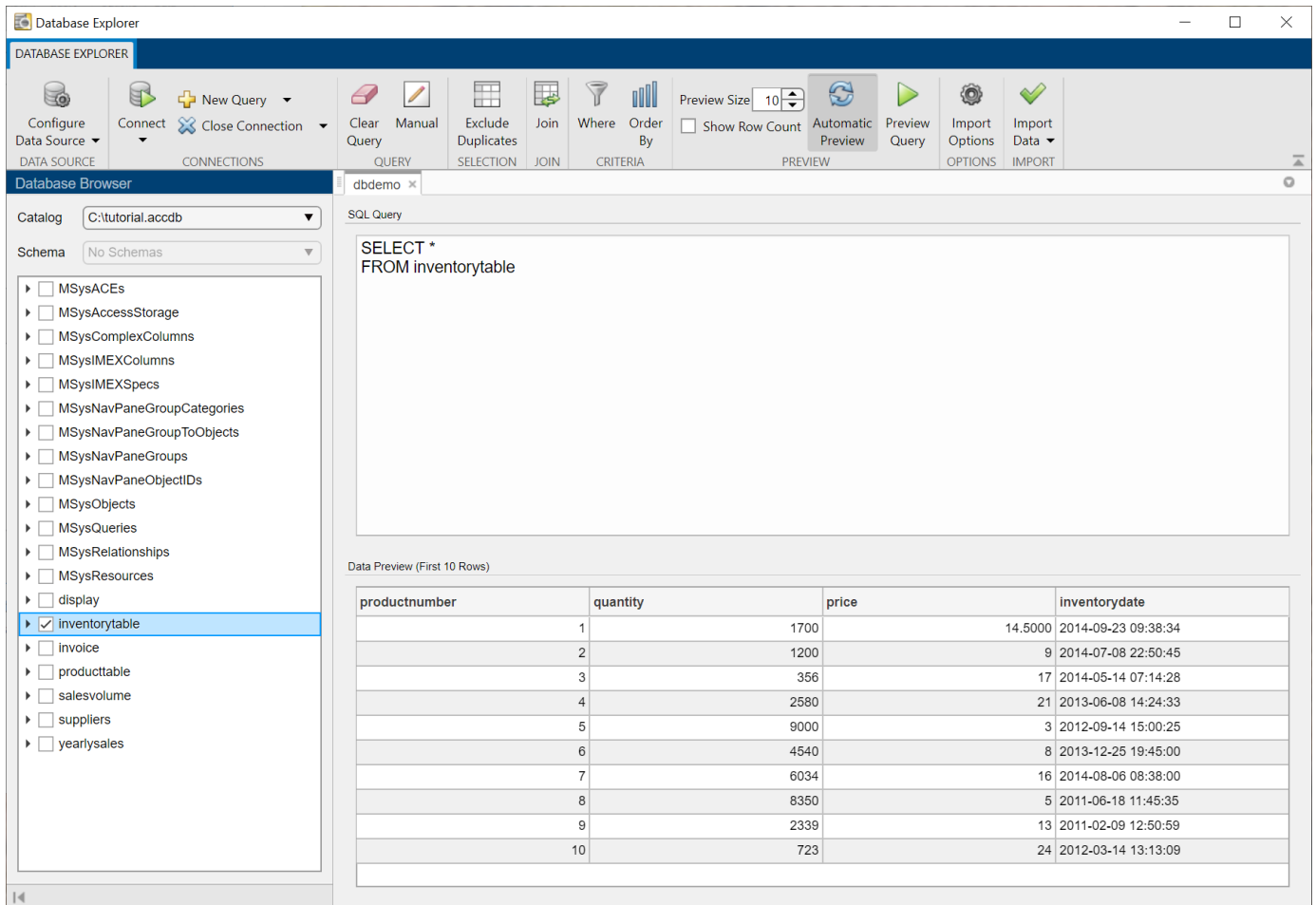
Using the **Database Explorer** app, you can open one or multiple database connections simultaneously by clicking **Connect** in the toolbar. The Database Explorer app creates a data source tab for each SQL query.

On each data source tab, you can write an SQL query in one of two ways. If you are unfamiliar with the SQL query language or want to explore data in your database, then use the **Data Browser** pane along with the buttons in the toolbar. Or, if you are already familiar with SQL, then enter an SQL query manually. When you enter a query, you can use more advanced SQL statements (for example, **AS**, **GROUP BY**, **HAVING**). You can also enter SQL code that is proprietary to the database and does not comply with the ANSI standard.

Create SQL Query Using Toolbar Buttons

Use these steps as a general workflow for creating an SQL query by using buttons in the toolbar.



- Connect to a data source by using the Database Explorer app. For an example, see “MySQL ODBC for Windows” on page 2-47.
- Click a table in the **Data Browser** pane. The **SQL Query** pane in the data source tab displays an SQL query that selects all columns and rows of the table. The **Data Preview** pane shows a preview of the first 10 rows of data in the table. For example, connect to a Microsoft Access database and select the `inventorytable` database table.



- In the **Join** section of the **Database Explorer** tab, click **Join** to display the **Join** tab in the toolbar. In the **Add** section of the **Join** tab, the name of the table selected in the **Data Browser** pane appears in the left **Table** list.
 - From the left **Column** list, select the name of the shared column.
 - From the right **Table** list, select the name of the table to join.
 - From the right **Column** list, select the name of the shared column.
 - Click **Add Join**. The app creates an inner join by default.
 - Close the **Join** tab.

For details about joining tables, see “Join Tables Using Database Explorer App” on page 4-10.

- In the **Data Browser** pane, expand the table name node of the joined table and select specific check boxes to choose the table columns. The **SQL Query** and **Data Preview** panes display the specified columns.
- In the **Criteria** section, click **Where** to display the **Where** tab in the toolbar. In the **Add** section, select an operator and value to enter an SQL **WHERE** condition. Click **Add Filter**. To represent strings in values, enclose text in single quotes. Close the **Where** tab.
- In the **Criteria** section, click **Order By** to display the **Order By** tab in the toolbar. In the **Add** section, select the column to sort and click **Add Sort**. Close the **Order By** tab.

-  In the **Import** section, click  to import all SQL query results into the MATLAB Workspace as a table.
- In the **Query** section, click **Clear Query** to clear the current SQL query and create a new one.
- In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

For detailed examples, see the **Database Explorer** app.


Enter SQL Query Manually


Use these steps as a general workflow for entering an SQL query manually.

- Connect to a data source in the Database Explorer app. For an example, see “MySQL ODBC for Windows” on page 2-47.
- In the **Query** section, click **Manual** to open a new data source tab. The tab has the same data source name as the prior active tab, but the Database Explorer app appends the suffix `_manual` to the tab name. The manual data source tab keeps the same database connection as the prior active tab.
- Enter or paste an SQL query into the **SQL Query** pane.

Note If you click **Manual** when the active data source tab contains an SQL query in the **SQL Query** pane, then you can modify the existing SQL query manually. Or, you can click **Clear Query** to clear the existing SQL query in the new tab and enter a new query.

For each subsequent time you click **Manual**, the new tab contains a numbered suffix.

- In the **Preview** section, click **Preview Query**. The Database Explorer app executes the SQL query and updates the **Data Preview** pane with the results. If the SQL query is valid, the **Data Preview** pane displays the first 10 rows of data by default. To see more rows, adjust the value in the **Preview Size** box.
- Modify the SQL query and click **Preview Query**. The **Data Preview** pane shows the updated results.
- 

In the **Import** section, click  to import all SQL query results into the MATLAB Workspace as a table.

- In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

For a detailed example of entering an SQL query manually, see the **Database Explorer** app.

Work with Multiple SQL Queries

To create different SQL queries using the same database, follow these steps:

- In the **Connections** section of the **Database Explorer** tab, click **Connect** and select the data source to create a database connection.
- In the connection dialog box, enter the user name and password for your database, and click **Connect**.
- In the Catalog and Schema dialog box, select the catalog and schema, and click **OK**. If only one catalog or schema is available in the database, the Catalog and Schema dialog box does not open.

The Database Explorer app opens a new tab with the data source name as the tab name.

- To create a different SQL query using the same database connection, click **New Query** in the **Connections** section.

The app opens a new data source tab and appends a numbered suffix to the tab name. The number increases by one in each subsequent data source tab name.

You can also create SQL queries in different catalogs or schemas. Repeat these steps and select a different catalog or schema in the Catalog and Schema dialog box.

To connect to a different database, repeat these steps and select a different data source from the **Connect** list.

SQL Query Limitations

The Database Explorer app has these limitations, which you can avoid by using the command line instead.

- You can connect to relational databases only.
- You can enter a single SQL **SELECT** statement only. You cannot enter other SQL statements or multiple SQL statements in the **SQL Query** pane.
- You cannot modify the database structure or the database data.
- You cannot execute stored procedures.

See Also

Functions

database

Apps

Database Explorer

More About

- “Configure Driver and Data Source” on page 2-14
- “Connect to Database” on page 2-121
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14

- “Generate SQL Query and MATLAB Script” on page 4-20
- “Database Explorer App Error Messages” on page 3-13

External Websites

- [SQL Tutorial](#)

Customize Import Options Using Database Explorer App

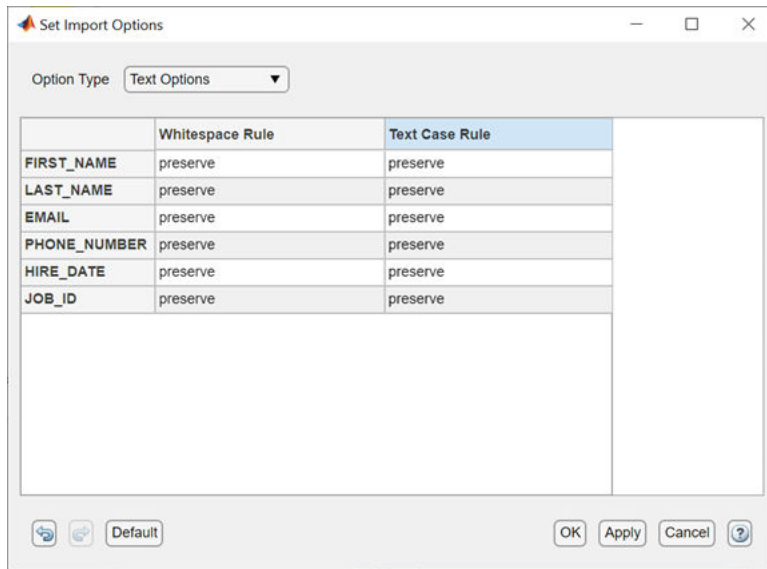
You can customize import options using the **Database Explorer** app after creating an SQL query and before importing data into the MATLAB Workspace. For details about creating SQL queries, see “Create SQL Queries Using Database Explorer App” on page 4-2.

Follow these steps as a general workflow to customize import options by using the Set Import Options dialog box. This workflow assumes that you create an SQL query that selects all data from a database table containing employee information.

- In the **Options** section, click **Import Options**. The Set Import Options dialog box opens and displays the default general options.

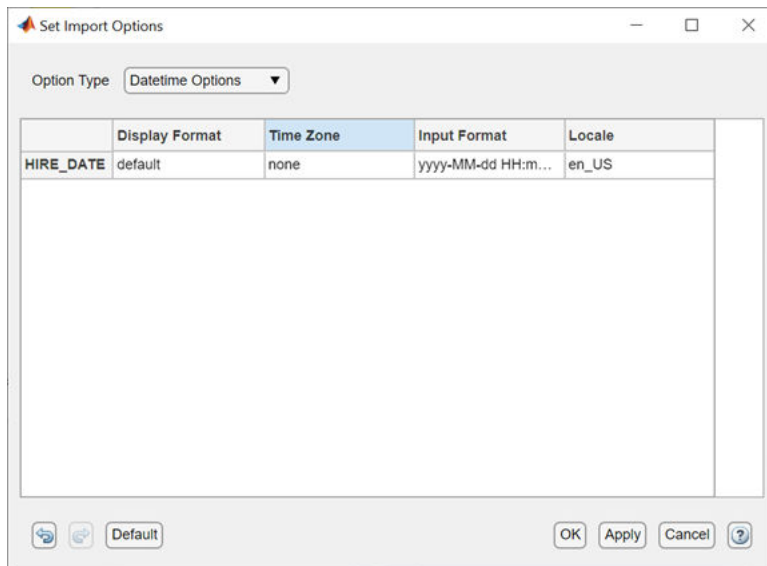
Variable Name	Variable Type	Missing Value
EMPLOYEE_ID	double	NaN
FIRST_NAME	char	
LAST_NAME	char	
EMAIL	char	
PHONE_NUMBER	char	
HIRE_DATE	char	
JOB_ID	char	
SALARY	double	NaN
COMMISSION_PCT	double	NaN
MANAGER_ID	double	NaN
DEPARTMENT_ID	double	NaN

- Under **Variable Name**, enter a different name for a specific variable to change its name in the imported data.
- Under **Variable Type**, select a different variable type for a specific variable to change its type in the imported data. The selected variable type must be compatible with the database type. For valid data types, see the `Option1,OptionValue1,...,OptionN,OptionValueN` input argument of the `setoptions` function.
- Under **Missing Value**, enter a different value for a specific variable to change how missing data is represented in the imported data. The data type of the specified value must match the variable type.
- From the **Option Type** list, select **Text Options**. The Set Import Options dialog box displays the import options for variables with text data types.



For details about customizing text import options, see the `Option1, OptionValue1, ..., OptionN, OptionValueN` input argument of the `setoptions` function.

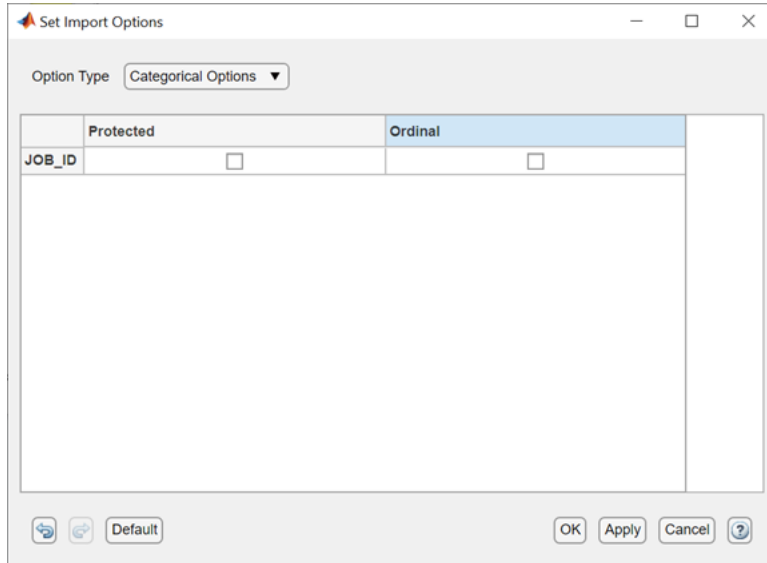
- After you select the `datetime` variable type for any variable in the general options, you can customize the datetime import options. For example, assume that you set the **HIRE_DATE** variable to the `datetime` variable type. From the **Option Type** list, select `Datetime Options`. The Set Import Options dialog box displays the import options for variables with `datetime` data types.



For details about customizing datetime import options, see the `Option1, OptionValue1, ..., OptionN, OptionValueN` input argument of the `setoptions` function.

- After you select the `categorical` variable type for any variable in the general options, you can customize the categorical import options. For example, assume that you set the **JOB_ID** variable

to the `categorical` variable type. From the **Option Type** list, select `Categorical Options`. The Set Import Options dialog box displays the import options for variables with `categorical` data types.




For details about customizing categorical import options, see the `Option1,OptionValue1,...,OptionN,OptionValueN` input argument of the `setoptions` function.

- Click **OK**. The dialog box closes and the **Data Preview** pane shows a preview of the data using the import options.

-



In the **Import** section, click  to import all SQL query results as a table in the MATLAB Workspace, according to the specified import options.

- In the **Import** section, select **Import Data > Generate MATLAB Script** to display a MATLAB script in the MATLAB Editor. The script mimics the actions of customizing import options in the Set Import Options dialog box for each option you set by using the `setoptions` function. For details about generating MATLAB scripts, see “Generate SQL Query and MATLAB Script” on page 4-20.

See Also

Apps
Database Explorer

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Generate SQL Query and MATLAB Script” on page 4-20

Join Tables Using Database Explorer App

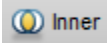

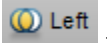

You can select and import data from multiple tables using the **Database Explorer** app. First, you must join tables, and then select the data to import. You can join tables using different join types that depend on the database.

Different Join Types

The Database Explorer app creates an inner join by default. To use another join type, click the corresponding button in the **Edit** section of the **Join** tab.



There are four join types:

-  **Inner** — An inner join retrieves records that have matching values in the selected column of both tables.
-  **Full** — A full join retrieves records that have matching values in the selected column of both tables, and unmatched records from both the left and right tables.
-  **Left** — A left join retrieves records that have matching values in the selected column of both tables, and unmatched records from the left table only.
-  **Right** — A right join retrieves records that have matching values in the selected column of both tables, and unmatched records from the right table only.

Join Tables

The Database Explorer app performs joins in one of two ways. The app can join tables automatically when you select tables by using shared columns (for example, the primary keys), or you can select tables without shared columns and manually specify the names of columns to match.

Automatic Join

The Database Explorer app can join tables automatically when you select tables in the **Data Browser** pane. In this case, the app checks if the selected tables have any column names in common. If there is a match, the app performs these steps.

- 1 Opens the **Join** tab.
- 2 Adds a join for each matched column name, creates an SQL query with the added joins, and executes the SQL query.
- 3 Displays the SQL query in the **SQL Query** pane and the query results in the **Data Preview** pane.

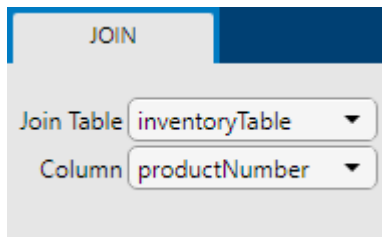
If the app does not find a match, the app displays an error dialog box that directs you to select a table in the **Join** tab. The app also removes the selections from the tables in the **Data Browser** pane.

Manual Join

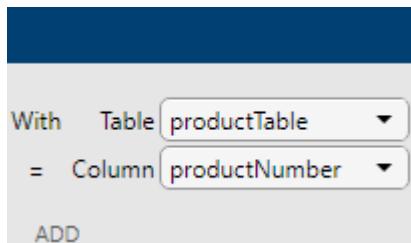
To join tables manually, you must know the names of each table and the names of the shared columns in the tables. Use these steps as a general workflow for joining tables.

- 1 After connecting to a database, select a table in the **Data Browser** pane. In the **Join** section, click **Join** to display the **Join** tab in the toolstrip. In the **Add** section, the name of the table selected in the **Data Browser** pane appears in the left **Table** list.

From the left **Column** list, select the name of the shared column.



- 2 From the right **Table** list, select the name of the table to join. From the right **Column** list, select the name of the shared column for this table.



- 3 In the **Add** section, click **Add Join**. The **SQL Query** pane updates the SQL query with the new join. If the **Automatic Preview** button (located in the **Preview** section of the **Database Explorer** tab) is toggled on, the **Data Preview** pane displays the updated SQL query results automatically. The **Join Diagram** pane displays a pictorial representation of the join between the selected tables.


The screenshot shows the 'JOIN' configuration window in Microsoft SQL Server Enterprise Manager. The 'Join Table' is set to 'inventorytable' and the 'With Table' is 'producttable'. The 'Column' is 'productnumber'. The 'INNER JOIN' is configured. The SQL Query pane shows the following query:

```
SELECT inventorytable.productnumber,
inventorytable.quantity,
inventorytable.price,
inventorytable.inventorydate
FROM ( inventorytable
INNER JOIN producttable
ON inventorytable.productnumber = producttable.productnumber)
```

The Data Preview pane shows the following data:

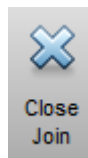
productnumber	quantity	price	inventorydate
9	2339	13	2011-02-09 12:50:59
8	8350	5	2011-06-18 11:45:35
7	6034	16	2014-08-06 08:38:00
2	1200	9	2014-07-08 22:50:45
4	2580	21	2013-06-08 14:24:33
1	1700	14.5000	2014-09-23 09:38:34
5	9000	3	2012-09-14 15:00:25
6	4540	8	2013-12-25 19:45:00
3	356	17	2014-05-14 07:14:28
10	723	24	2012-03-14 13:13:09

The Join Diagram pane shows a diagram with 'inventorytable' and 'producttable' connected by an 'INNER JOIN' node.

- 4 To add another join, select another table and column name combination in the left and right lists. Then, click **Add Join** again.
- 5 In the **Edit** section, click one of the join types (for example, ) to specify a different join type, if necessary.
- 6 To remove a join, select it in the list of joins in the **Edit** section, and click **Remove Join**.

Note To change the order of joins, remove existing joins and create joins in another order.

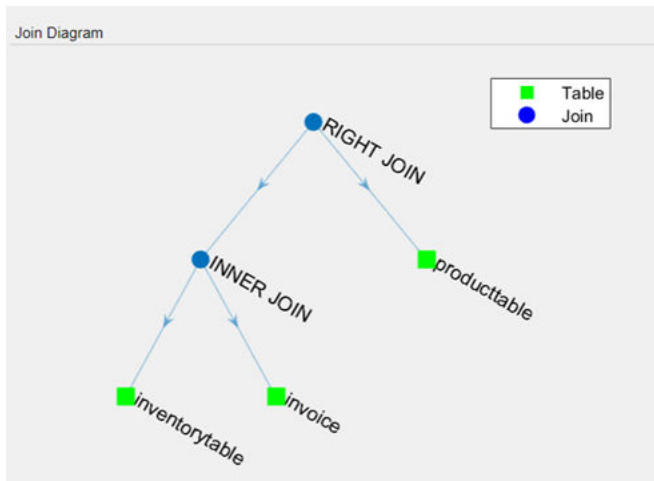
7



In the **Close** section, click **Close Join** to close the **Join** tab.

Join Diagram

After you join at least two tables, the **Join Diagram** pane displays a pictorial representation of the joins between tables. Each blue circle shows the join type. Each green square shows a table in the join.



When working with multiple joins, use this diagram to see the hierarchy of joins. Ensure that you are using the correct join types for your data. As you modify join types, the diagram updates to reflect the new join types.

Join Type Limitations

Some database vendors do not support all join types. The Database Explorer app enables the corresponding buttons in the **Join** tab for the supported join types in these databases:

- SQLite supports only inner and left join types.
- Microsoft Access and MySQL support only inner, left, and right join types.

See Also

Apps
Database Explorer

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Data Preview Using Database Explorer App” on page 4-14
- “Generate SQL Query and MATLAB Script” on page 4-20
- “Database Explorer App Error Messages” on page 3-13

External Websites

- SQL Tutorial

Data Preview Using Database Explorer App

Using the **Database Explorer** app, you can preview data in the **Data Preview** pane when you:

- Select tables and columns in the **Data Browser** pane.
- Create an SQL query using the buttons in the toolbar.
- Enter an SQL query manually.

After previewing the data, you can modify the SQL query or import the data into the MATLAB Workspace.

Automatic Preview

The Database Explorer app previews data automatically, by default. With the **Automatic Preview** button toggled on in the **Preview** section and a valid SQL query in the **SQL Query** pane, the Database Explorer app executes the SQL query and previews the data in the **Data Preview** pane. You can control the number of rows displayed in this pane by entering a value in the **Preview Size** box in the toolbar.

As you create an SQL query, the **Data Preview** pane updates the preview of the query results for each change you make to the query. When you change selections in the **Data Browser** pane, or add or remove a join, filter, or sort, the **SQL Query** pane updates along with the **Data Preview** pane.

If your SQL query takes a long time to execute or you know that the query results contain a large amount of data, you can stop the automatic preview of data. Click the **Automatic Preview** button in the **Preview** section of the toolbar. When this button is toggled off, you can modify the query in the **SQL Query** pane without updating the preview of data. After you finish modifying the query, click **Preview Query** in the **Preview** section to see the updated query results in the **Data Preview** pane.

Preview Size

To see more rows in the **Data Preview** pane, enter a larger value in the **Preview Size** box in the **Preview** section. The number of rows in the **Data Preview** pane matches the entered value. However, if the number of rows in the SQL query results is less than the entered value, then all rows are displayed in the **Data Preview** pane.

To see the count of the total number of rows in the SQL query results, select the **Show Row Count** check box in the **Preview** section. The **Data Preview** pane displays the row count of the SQL query results. For example, using the sample database connection:

- 1 In the **Data Browser** pane, select **inventorytable**. The **SQL Query** pane displays the SQL query that selects all data from the database table **inventorytable**.
- 2 In the **Preview** section, select the **Show Row Count** check box. The **Data Preview** pane displays the row count **13** in the note within parentheses.

The screenshot shows the Database Explorer application interface. The top toolbar includes buttons for 'Configure Data Source', 'Connect', 'Close Connection', 'Clear Query', 'Manual Query', 'Exclude Duplicates', 'Join', 'Where', 'Order By', 'Preview Size' (set to 10), 'Show Row Count' (checked), 'Automatic Preview', 'Preview Query', 'Import Options', and 'Import Data'. The 'Database Browser' pane on the left shows a tree view of the 'C:\tutorial.accdb' database with various system tables and user tables. The 'inventorytable' is selected. The 'SQL Query' pane contains the query: `SELECT * FROM inventorytable`. The 'Data Preview (First 10 of 13 Rows)' pane displays the following data:

productnumber	quantity	price	inventorydate
1	1700	14.5000	2014-09-23 09:38:34
2	1200	9	2014-07-08 22:50:45
3	356	17	2014-05-14 07:14:28
4	2580	21	2013-06-08 14:24:33
5	9000	3	2012-09-14 15:00:25
6	4540	8	2013-12-25 19:45:00
7	6034	16	2014-08-06 08:38:00
8	8350	5	2011-06-18 11:45:35
9	2339	13	2011-02-09 12:50:59
10	723	24	2012-03-14 13:13:09

Preview Data by Creating SQL Query

After you connect to a database, the tables in the database are displayed in the **Data Browser** pane. Use this pane along with buttons in the toolbar to view the database structure and create an SQL query.

- Expand and collapse the tables to see the columns in each table.
- Click the box next to a table name to see a quick preview of the data in the table. If the **Automatic Preview** button is toggled on, the **Data Preview** pane displays the first 10 rows of data automatically.
- Click boxes next to column names to see a quick preview of data in those columns only.
- Use the buttons in the toolbar to exclude duplicate rows, join tables, filter data, and sort data.

After you create a valid SQL query in the **SQL Query** pane, the Database Explorer app executes the query and previews the results as described earlier in Automatic Preview.

Preview Data by Entering SQL Query Manually

You can enter an SQL query manually or by pasting text into the **SQL Query** pane. Or, you can modify an existing SQL query in the **SQL Query** pane. To enter or modify an SQL query, click **Manual** in the **Edit** section.

Note When you enter or modify an SQL query manually, the Database Explorer app stops the automatic preview of data.

After you enter the SQL query, click **Preview Query** in the **Preview** section. The **Data Preview** pane displays the first 10 rows of data only after you click this button. If you modify the SQL query, then click **Preview Query** again to refresh the SQL query results.

See Also

Apps
Database Explorer

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Generate SQL Query and MATLAB Script” on page 4-20
- “Database Explorer App Error Messages” on page 3-13

Modify and Delete Data Sources

You can modify and delete ODBC and JDBC data sources using the **Database Explorer** app or the command line. For ODBC drivers, you can create, modify, and delete data sources using the ODBC Data Source Administrator dialog box. For JDBC drivers, you can create, modify, and delete data sources using the JDBC Data Source Configuration dialog box or the command line.

Modify Data Sources Interactively

Use the Database Explorer app to modify data sources by following these steps.

ODBC Data Sources

- 1 Open the Database Explorer app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure ODBC data source**. The ODBC Data Source Administrator dialog box opens.

Alternatively, run the `configureODBCDataSource` function.

- 2 In the ODBC Data Source Administrator dialog box, select the data source to modify. Click **Configure**.
- 3 Modify the settings as needed.

JDBC Data Sources

- 1 Opening the Database Explorer app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure JDBC data source**. The JDBC Data Source Configuration dialog box opens.
- 2 Click **Edit**. In the list, select the data source to modify. Click **OK**.
- 3 Modify the settings in the JDBC Data Source Configuration dialog box. If you do not change the data source name, the Database Explorer app overwrites the existing data source with the new settings. To avoid overwriting the existing data source, enter a new data source name.
- 4 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database. Click **Test**.

If your connection succeeds, the app displays a message indicating a successful connection. Otherwise, it displays an error message.

- 5 Click **Save**.

Modify Data Sources Programmatically

Use the command line to modify data sources by following these steps.

ODBC Data Sources

Modify an ODBC data source using the ODBC Data Source Administrator dialog box. To access this dialog box, see the `configureODBCDataSource` function.

JDBC Data Sources

- 1 Edit an existing JDBC data source using the `databaseConnectionOptions` function.

- 2 Set JDBC connection options using the `setoptions` function. Or, you can set properties in the `SQLConnectionOptions` object using dot notation.
- 3 Add JDBC driver-specific connection options using the `setoptions` function. Or, remove existing JDBC driver-specific options using the `rmoptions` function.
- 4 Test the database connection using the `testConnection` function.
- 5 Save the JDBC data source using the `saveAsDataSource` function.

Delete Data Sources Interactively

Use the Database Explorer app to delete data sources by following these steps.

ODBC Data Sources

- 1 Open the Database Explorer app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure ODBC data source**. The ODBC Data Source Administrator dialog box opens.
- 2 Select the data source to delete.
- 3 Click **Remove**.

JDBC Data Sources

- 1 Open the Database Explorer app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure JDBC data source**. The JDBC Data Source Configuration dialog box opens.
- 2 Click **Edit**.
- 3 In the list, select the name of the data source to delete. Click **OK**.
- 4 Click **Delete** and click **Yes**.

Delete Data Sources Programmatically

Delete an ODBC data source using the ODBC Data Source Administrator dialog box. To access this dialog box, see the `configureODBCDataSource` function.

To delete a JDBC data source, specify the name of the data source in the `deleteDataSource` function.

See Also

Apps

Database Explorer

Objects

`SQLConnectionOptions`

Functions

`databaseConnectionOptions` | `setoptions` | `rmoptions` | `testConnection` | `saveAsDataSource` | `deleteDataSource`

More About

- “Choose Between ODBC and JDBC Drivers” on page 2-12
- “Configure Driver and Data Source” on page 2-14
- “Create JDBC Data Source and Set Options Programmatically” on page 2-17
- “Connect to Database” on page 2-121

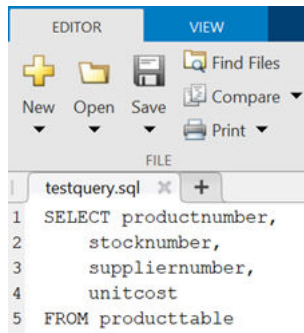
Generate SQL Query and MATLAB Script

You can generate SQL code from an SQL query or create a MATLAB script by using the **Database Explorer** app. After defining an SQL query in the **SQL Query** pane, you can generate the SQL code for running an SQL script. You can also generate a MATLAB script to automate connecting to a database, running an SQL query, and performing data analysis on the imported data.

Generate SQL Query

To generate SQL code from an SQL query, follow these steps:

- 1 Connect to a data source and create an SQL query. For details about creating SQL queries, see “Create SQL Queries Using Database Explorer App” on page 4-2.
- 2 In the **Import** section, select **Import Data > Generate SQL Query**.
- 3 Save the SQL code to a .txt or .sql file. The MATLAB Editor opens the saved SQL code file.



You can use the SQL code to rebuild a query by using the **SQL Query** pane by entering the SQL code manually.

Alternatively, you can use the .sql file to import data programmatically into MATLAB by using the `executeSQLScript` function.

Generate MATLAB Script

To generate a MATLAB script that automates connecting to a data source, running an SQL query, and importing the SQL query results, follow these steps:

- 1 Connect to a data source and create an SQL query. For details about creating SQL queries, see “Create SQL Queries Using Database Explorer App” on page 4-2.
- 2 In the **Import** section, select **Import Data > Generate MATLAB Script** to display a MATLAB script in the MATLAB Editor. To customize import options before importing data, see “Customize Import Options Using Database Explorer App” on page 4-7.

The screenshot shows the MATLAB Editor window with a script titled 'Untitled*'. The script contains the following MATLAB code:

```

1  %% Automate Importing Data by Generating Code Using the Database Explorer App
2  % This code reproduces the data obtained using the Database Explorer app by
3  % connecting to a database, executing a SQL query, and importing data into
4  % the MATLAB(R) workspace. To use this code, add the password for
5  % connecting to the database in the database command.
6
7  % Auto-generated by MATLAB Version 9.11 (R2021b) and Database Toolbox
8  % Version 10.0 on 09-Dec-2020 15:27:12
9
10 %% Make connection to database
11 conn = database('MS SQL Server Auth','','');
12
13 %Set query to execute on the database
14 query = ['SELECT productNumber, ' ...
15         '   Quantity, ' ...
16         '   Price ' ...
17         'FROM toystore_doc.dbo.inventoryTable'];
18
19 %% Execute query and fetch results
20 data = fetch(conn,query);
21
22 %% Close connection to database
23 close(conn)
24
25 %% Clear variables
26 clear conn query

```

- 3 Save the MATLAB script. You can run this script at the command line.

Note To run the script, enter the database password as an input argument of the `database` function in the script.

For details about editing and running scripts, see “Scripts”.

See Also

Functions

`executeSQLScript`

Apps

`Database Explorer`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Customize Import Options Using Database Explorer App” on page 4-7

Using Database Toolbox Functions

- “Roll Back Data in Database” on page 5-2
- “Change Database Connection Catalog” on page 5-4
- “Create Table and Add Column” on page 5-5
- “Delete Data from Databases” on page 5-6
- “Roll Back Data After Updating Record” on page 5-9
- “Replace Existing Data in Database” on page 5-12
- “Export Data Using Bulk Insert” on page 5-13
- “Call Stored Procedure That Returns Data” on page 5-17
- “Run Custom Database Function” on page 5-19
- “Data Import Memory Management” on page 5-20
- “Import Large Data Using DatabaseDatastore Object” on page 5-23
- “Analyze Large Data in Database Using MapReduce” on page 5-27
- “Analyze Large Data in Database Using Tall Arrays” on page 5-30
- “Import Data Using MATLAB Interface to SQLite” on page 5-32
- “Insert Data into SQLite Database Table” on page 5-36
- “Create Table and Add Column in SQLite Database” on page 5-38
- “Delete Data from SQLite Database” on page 5-39
- “Roll Back Data in SQLite Database” on page 5-41
- “Deploy MATLAB Interface to SQLite Database Application with MATLAB Compiler” on page 5-43
- “Retrieve Image Data Types” on page 5-48
- “Import Boolean Data from Database” on page 5-51
- “Append Data to Existing Database Table Using Insert Functionality” on page 5-53
- “Insert Data into New Database Table Using Insert Functionality” on page 5-55
- “Join Tables Using Command Line” on page 5-57
- “Import Data from Database Table Using sqlread Function” on page 5-58
- “Insert Data into Database Table” on page 5-61
- “Retrieve Database Metadata” on page 5-64
- “Customize Options for Importing Data from Database into MATLAB” on page 5-67
- “Deploy Relational Database Application with MATLAB Compiler” on page 5-70
- “Import Data Using SQL Prepared Statement with Multiple Parameter Values” on page 5-75

Roll Back Data in Database

This example shows how to connect to a database, update an existing row of data in the database, and roll back the update. Use the `execute` function to roll back the update after executing the `update` function.

Create a database connection `conn` to the Microsoft Access database. This code assumes that you are connecting to a data source named `dbdemo` with blank user name and password.

```
conn = database('dbdemo', '', '');
```

This database contains the table `inventorytable` that contains these columns:

- `productnumber`
- `quantity`
- `price`
- `inventorydate`

Set the `AutoCommit` property of the connection object to `'off'`. Any updates you make after turning off this flag do not commit to the database automatically.

```
conn.AutoCommit = 'off';
```

Define a cell array containing the column names that you are updating in `inventorytable`.

```
colnames = {'price', 'inventorydate'};
```

Define a table that contains the data for insertion. Update the price to \$15 and set the inventory timestamp to `'2014-12-01 08:50:15.000'`.

```
data = table(15, {'2014-12-01 08:50:15.000'}, ...  
    'VariableNames', {'price', 'inventorydate'});
```

Update the columns `price` and `inventorydate` in the table `inventorytable` for the product number equal to 1.

```
tablename = 'inventorytable';  
whereclause = 'WHERE productnumber = 1';
```

```
update(conn, tablename, colnames, data, whereclause)
```

Roll back data for the update.

```
sqlquery = 'ROLLBACK';  
execute(conn, sqlquery)
```

You can commit data to the database by replacing the `ROLLBACK` SQL statement with `COMMIT`. You can also roll back or commit data after executing an `INSERT` SQL statement using the `sqlwrite` function.

Close the database connection.

```
close(conn)
```

See Also

`execute` | `close` | `database`

Related Examples

- “Insert Data into Database Table” on page 5-61
- “Replace Existing Data in Database” on page 5-12

Change Database Connection Catalog

This example shows how to connect to a database and change the database catalog. You can work with data in a different catalog within the same database using the `execute` function.

Create an ODBC database connection to a Microsoft SQL Server database with Windows authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Change the catalog for the database connection `conn` to `intlprice` by creating and executing an SQL query.

```
sqlquery = 'Use intlprice';  
execute(conn, sqlquery)
```

Close the database connection.

```
close(conn)
```

See Also

[execute](#) | [close](#) | [database](#)

Related Examples

- “Retrieve Database Metadata” on page 5-64

External Websites

- [SQL Tutorial](#)

Create Table and Add Column

This example shows how to connect to a database and manage the database structure. You can manage the database structure using the `execute` function.

Create an ODBC database connection to a Microsoft SQL Server database with Windows authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Use the SQL `CREATE` statement to create the table `Person`.

```
sqlquery = ['CREATE TABLE Person(LastName varchar, ' ...  
           'FirstName varchar,Address varchar,Age int)'];
```

Create the table in the database using the database connection.

```
execute(conn,sqlquery)
```

Use the SQL `ALTER` statement to add the column `City` to the table `Person`.

```
sqlquery = 'ALTER TABLE Person ADD City varchar(30)';  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

See Also

[execute](#) | [close](#) | [database](#)

Related Examples

- “Retrieve Database Metadata” on page 5-64

External Websites

- [SQL Tutorial](#)

Delete Data from Databases

This example shows how to delete data from your database using MATLAB.

Create the SQL statement with your deletion SQL syntax. Consult your database documentation for the correct SQL syntax. Execute the delete operation on your database using the `execute` function with your SQL statement. This example demonstrates deleting data records in a Microsoft Access database.

Connect to Database

Create the database connection `conn` to a Microsoft Access database using an ODBC driver and the data source name `dbdemo`. This database contains the table `inventorytable` with the column `productnumber`.

```
conn = database('dbdemo', '', '');
```

The SQL query `sqlquery` selects all rows of data in the table `inventorytable`. Execute this SQL query using `conn`. Import the data from the executed query using the `fetch` function and display the last few rows.

```
sqlquery = 'SELECT * FROM inventorytable';
data = fetch(conn, sqlquery);
tail(data)
```

```
ans =
```

```
8×4 table
```

productnumber	quantity	price	inventorydate
6	4540	8	'2013-12-25 19:45:00'
7	6034	16	'2014-08-06 08:38:00'
8	8350	5	'2011-06-18 11:45:35'
9	2339	13	'2011-02-09 12:50:59'
10	723	24	'2012-03-14 13:13:09'
11	567	0	'2012-09-11 00:30:24'
12	1278	0	'2010-10-29 18:17:47'
13	1700	14.5	'2009-05-24 10:58:59'

Delete Specific Record

Delete the data for the product number 13 from the table `inventorytable`. Specify the product number using the `WHERE` clause in the SQL statement `sqlquery`.

```
sqlquery = 'DELETE * FROM inventorytable WHERE productnumber = 13';
execute(conn, sqlquery)
```

Display the data in the table `inventorytable` after the deletion. The record with product number 13 is missing.

```
sqlquery = 'SELECT * FROM inventorytable';
data = fetch(conn, sqlquery);
tail(data)
```

```
ans =
```


8×4 table

productnumber	quantity	price	inventorydate
5	9000	3	'2012-09-14 15:00:25'
6	4540	8	'2013-12-25 19:45:00'
7	6034	16	'2014-08-06 08:38:00'
8	8350	5	'2011-06-18 11:45:35'
9	2339	13	'2011-02-09 12:50:59'
10	723	24	'2012-03-14 13:13:09'
11	567	0	'2012-09-11 00:30:24'
12	1278	0	'2010-10-29 18:17:47'

Delete Record Using MATLAB Variable

Define a MATLAB variable `productID` by setting it to the product number 12.

```
productID = 12;
```

Delete the data using the MATLAB variable `productID`. Build an SQL statement `sqlquery` that combines the SQL for the delete operation with the MATLAB variable. Since the variable is numeric and the SQL statement is a character vector, convert the number to a character vector. Use the `num2str` function for the conversion. Concatenate the delete SQL statement and the numeric conversion using the square brackets.

```
sqlquery = ['DELETE * FROM inventorytable WHERE ' ...
            'productnumber = ' num2str(productID)];
execute(conn,sqlquery)
```

Display the data in the table `inventorytable` after the deletion. The record with product number 12 is missing.

```
sqlquery = 'SELECT * FROM inventorytable';
data = fetch(conn,sqlquery);
tail(data)
```

```
ans =
```

8×4 table

productnumber	quantity	price	inventorydate
4	2580	21	'2013-06-08 14:24:33'
5	9000	3	'2012-09-14 15:00:25'
6	4540	8	'2013-12-25 19:45:00'
7	6034	16	'2014-08-06 08:38:00'
8	8350	5	'2011-06-18 11:45:35'
9	2339	13	'2011-02-09 12:50:59'
10	723	24	'2012-03-14 13:13:09'
11	567	0	'2012-09-11 00:30:24'

Close Database Connection

`close(conn)`

See Also

`execute` | `fetch` | `num2str`

Related Examples

- “Import Data from Database Table Using `sqlread` Function” on page 5-58

External Websites

- [SQL Tutorial](#)

Roll Back Data After Updating Record

This example shows how to update data in a database and roll back the changes. Rolling back the changes reinstates the data as it appears before running the update.

Create a database connection `conn`. For example, the following code uses the database `toy_store`, user name `username`, password `pwd`, server name `sname`, and port number `123456` to connect to a Microsoft SQL Server database. This database contains the table `inventoryTable` that contains these columns: `productNumber`, `Quantity`, and `Price`.

```
conn = database('toy_store', 'username', 'pwd', ...
               'Vendor', 'Microsoft SQL Server', ...
               'Server', 'sname', ...
               'PortNumber', 123456);
```

Set the `AutoCommit` property of the connection object to `'off'`. Any updates you make after turning off this flag do not commit to the database automatically.

```
conn.AutoCommit = 'off';
```

Display the data in the `inventoryTable` table before making updates. Import the data from the executed query using the `fetch` function and display the first few rows of imported data.

```
d = fetch(conn, 'SELECT * FROM inventoryTable');
head(d)
```

```
ans =
```

```
8×4 table
```

<u>productnumber</u>	<u>quantity</u>	<u>price</u>	<u>inventorydate</u>
1	1700	14.5	'2014-09-23 09:38:34'
2	1200	9	'2014-07-08 22:50:45'
3	356	17	'2014-05-14 07:14:28'
4	2580	21	'2013-06-08 14:24:33'
5	9000	3	'2012-09-14 15:00:25'
6	4540	8	'2013-12-25 19:45:00'
7	6034	16	'2014-08-06 08:38:00'
8	8350	5	'2011-06-18 11:45:35'

Define a cell array for the new price of the first product.

```
data(1,1) = {30.00};
```

Define the `WHERE` clause for the first product.

```
whereclause = 'where productNumber = 1';
```

Update the `Price` column in the `inventoryTable` for the first product.

```
tablename = 'inventoryTable';
colname = {'Price'};
```

```
update(conn, tablename, colname, data, whereclause)
```

Display the data in the `inventoryTable` table after making the update.

```
d = fetch(conn, 'SELECT * FROM inventoryTable');
head(d)
```

```
ans =
```

```
8×4 table
```

productnumber	quantity	price	inventorydate
1	1700	30	'2014-09-23 09:38:34'
2	1200	9	'2014-07-08 22:50:45'
3	356	17	'2014-05-14 07:14:28'
4	2580	21	'2013-06-08 14:24:33'
5	9000	3	'2012-09-14 15:00:25'
6	4540	8	'2013-12-25 19:45:00'
7	6034	16	'2014-08-06 08:38:00'
8	8350	5	'2011-06-18 11:45:35'

The first product has an updated price of 30. Though the data is updated, the change has not committed to the database.

Roll back the update.

```
rollback(conn)
```

Alternatively, you can roll back the update using an SQL ROLLBACK statement by using the `execute` function.

Display the data in the `inventoryTable` table after rolling back the update.

```
d = fetch(conn, 'SELECT * FROM inventoryTable');
head(d)
```

```
ans =
```

```
8×4 table
```

productnumber	quantity	price	inventorydate
1	1700	14.5	'2014-09-23 09:38:34'
2	1200	9	'2014-07-08 22:50:45'
3	356	17	'2014-05-14 07:14:28'
4	2580	21	'2013-06-08 14:24:33'
5	9000	3	'2012-09-14 15:00:25'
6	4540	8	'2013-12-25 19:45:00'
7	6034	16	'2014-08-06 08:38:00'
8	8350	5	'2011-06-18 11:45:35'

The first product has the old price of 14.50.

Close the database connection.

```
close(conn)
```

See Also

`execute` | `database` | `fetch` | `rollback` | `close`

Related Examples

- “Insert Data into Database Table” on page 5-61
- “Replace Existing Data in Database” on page 5-12

External Websites

- [SQL Tutorial](#)

Replace Existing Data in Database

This example shows how to update a value of the month column in the table `yearlysales` using the data source named `dbdemo`. To access the example where you import the values of the month column, see “Insert Data into Database Table” on page 5-61.

Create a database connection `conn` to the Microsoft Access database using the ODBC driver. Here, this code assumes that you are connecting to a data source named `dbdemo` with blank user name and password.

```
conn = database('dbdemo', '', '');
```

To update the month, specify the month column that contains the months in the cell array `colnames`.

```
colnames = {'month'};
```

Assign the month value `March2010` to the MATLAB variable `data` for the update. The data type of `data` is a table.

```
data = table({'March2010'}, 'VariableNames', {'month'});
```

Specify the record to update in the database by defining an SQL WHERE statement `whereclause`. The record to update is the record whose month is `March`. Embed `March` in two single quotation marks so that MATLAB interprets `March` as a character vector in the SQL WHERE statement.

```
whereclause = 'WHERE month = 'March''
```

```
whereclause =
```

```
    'WHERE month = 'March''
```

Update the data for the record whose month is `March` in the database table `yearlysales`.

```
update(conn, 'yearlysales', colnames, data, whereclause)
```

In Microsoft Access, view the `yearlysales` table to verify the results.

month	salestotal	revenue
March2010	14606	\$0.00

Close the database connection.

```
close(conn)
```

See Also

[update](#) | [database](#) | [close](#)

Related Examples

- “Insert Data into Database Table” on page 5-61

External Websites

- [SQL Tutorial](#)

Export Data Using Bulk Insert

Bulk Insert Functionality

One way to export data from MATLAB and insert it into your database is to use the `sqlwrite` function at the command line. However, if you experience performance issues with this process, you can instead create a data file containing every record in your data set. Then, you can use this data file as input into the bulk insert functionality of your database to process the large data set. Also, with this file, you can insert data with special characters such as double quotes. A bulk insert provides performance gains by using the bulk insert utilities that are native to different database systems. For details, see “Working with Large Data Sets” on page 2-127.

The following examples use preconfigured JDBC data sources. For more information about configuring a JDBC data source, see the `databaseConnectionOptions` function.

Bulk Insert into Oracle

This example uses a data file containing sports data on a local machine with Oracle installed and exports data in the file to the Oracle server using bulk insert functionality.

- 1 Connect to a configured JDBC data source for the Oracle database.

```
datasource = "ORA_JDBC";
username = "user";
password = "password";
conn = database(datasource,username,password);
```

- 2 Create a table named BULKTEST using the `execute` function.

```
execute(conn,['CREATE TABLE BULKTEST (salary number, ' ...
            'player varchar2(25), signed varchar2(25), ' ...
            'team varchar2(25))'])
```

- 3 Create a data record.

```
A = {100000.00, 'KGreen', '06/22/2011', 'Challengers'};
```

- 4 Expand A to a data set containing 10,000 records.

```
A = A(ones(10000,1),:);
```

- 5 Write data to a file for bulk insert functionality.

Tip When connecting to a database on a remote machine, you must write this file to the remote machine. Oracle has difficulty reading files that are not on the same machine as the database.

```
fid = fopen('c:\temp\tmp.txt','wt');
for i = 1:size(A,1)
    fprintf(fid,'%10.2f \t %s \t %s \t %s \n',A{i,1}, ...
            A{i,2},A{i,3},A{i,4});
end
fclose(fid);
```

- 6 Set the folder location using the `execute` function.

- ```
execute(conn, ...
 'CREATE OR REPLACE DIRECTORY ext AS 'C:\\Temp''')
7 Delete the temporary table, if it exists, using the execute function.
execute(conn, 'DROP TABLE testinsert')
8 Create a temporary table and use bulk insert functionality to insert it into the table BULKTEST.
execute(conn, ['CREATE TABLE testinsert (salary number, ' ...
 'player varchar2(25), signed varchar2(25), ' ...
 'team varchar2(25)) ORGANIZATION EXTERNAL ' ...
 '(TYPE ORACLE_LOADER DEFAULT DIRECTORY ext ACCESS ' ...
 'PARAMETERS (RECORDS DELIMITED BY NEWLINE FIELDS ' ...
 'TERMINATED BY '\\t') LOCATION ('tmp.txt')) ' ...
 'REJECT LIMIT 10000'])
execute(conn, 'INSERT INTO BULKTEST SELECT * FROM testinsert')
9 Confirm the number of variables in BULKTEST.
results = fetch(conn, 'SELECT * FROM BULKTEST');
results.Properties.VariableNames
ans =
 1x4 cell array
 {'SALARY'} {'PLAYER'} {'SIGNED'} {'TEAM'}
```
- 10 Close the database connection.
- ```
close(conn)
```

Bulk Insert into Microsoft SQL Server 2005

This example uses a data file containing sports data on a local machine with Microsoft SQL Server installed and exports data in the file to the Microsoft SQL Server using bulk insert functionality.

- 1 Connect to a configured JDBC data source for the Microsoft SQL Server database.

```
datasource = "MSSQLServer_JDBC";
username = "user";
password = "password";
conn = database(datasource, username, password);
```

- 2 Create a table named BULKTEST using the execute function.

```
execute(conn, ['CREATE TABLE BULKTEST (salary ' ...
    'decimal(10,2), player varchar(25), signed_date ' ...
    'datetime, team varchar(25))'])
```

- 3 Create a data record.

```
A = {100000.00, 'KGreen', '06/22/2011', 'Challengers'};
```

- 4 Expand A to a data set containing 10,000 records.

```
A = A(ones(10000, 1), :);
```

- 5 Write data to a file for bulk insert functionality.

Tip When connecting to a database on a remote machine, you must write this file to the remote machine. Microsoft SQL Server has difficulty reading files that are not on the same machine as the database.

```

fid = fopen('c:\temp\tmp.txt','wt');
for i = 1:size(A,1)
    fprintf(fid,'%10.2f \t %s \t %s \t %s \n',A{i,1}, ...
        A{i,2},A{i,3},A{i,4});
end
fclose(fid);

```

- 6 Run the bulk insert functionality using the `execute` function.

```

execute(conn,['BULK INSERT BULKTEST FROM ' ...
    ''c:\temp\tmp.txt'' WITH (FIELDTERMINATOR = ''\t'', ' ...
    'ROWTERMINATOR = ''\n'')'])

```

- 7 Confirm the number of variables in BULKTEST.

```

results = fetch(conn,'SELECT * FROM BULKTEST');
results.Properties.VariableNames

```

```
ans =
```

```
1x4 cell array
```

```
{'SALARY'} {'PLAYER'} {'SIGNED_DATE'} {'TEAM'}
```

- 8 Close the database connection.

```
close(conn)
```

Bulk Insert into MySQL

This example uses a data file containing sports data on a local machine with MySQL installed and exports data in the file to a MySQL database using bulk insert functionality.

- 1 Connect to a configured JDBC data source for the MySQL database.

```

datasource = "MySQL_JDBC";
username = "user";
password = "password";
conn = database(datasource,username,password);

```

- 2 Create a table named BULKTEST using the `execute` function.

```

execute(conn,['CREATE TABLE BULKTEST (salary decimal, ' ...
    'player varchar(25), signed_date varchar(25), ' ...
    'team varchar(25)'])

```

- 3 Create a data record.

```
A = {100000.00, 'KGreen', '06/22/2011', 'Challengers'};
```

- 4 Expand A to a data set containing 10,000 records.

```
A = A(ones(10000,1),:);
```

- 5 Write data to a file for bulk insert functionality.

Note MySQL reads files saved locally, even if you are connecting to a remote machine. Therefore, you can write the file to either your local or remote machine.

```

fid = fopen('c:\temp\tmp.txt','wt');
for i = 1:size(A,1)

```

```
fprintf(fid,'%10.2f \t %s \t %s \t %s \n', ...  
A{i,1},A{i,2},A{i,3},A{i,4});  
end  
fclose(fid);
```

- 6 Run the bulk insert functionality. The SQL statement uses the statement `LOCAL INFILE` for error handling. For details about this statement, consult the MySQL database documentation.

```
execute(conn,['LOAD DATA LOCAL INFILE ' ...  
' 'C:\\temp\\tmp.txt' INTO TABLE BULKTEST ' ...  
' FIELDS TERMINATED BY '\\t' LINES TERMINATED ' ...  
' BY '\\n'''])
```

- 7 Confirm the number of variables in `BULKTEST`.

```
results = fetch(conn,'SELECT * FROM BULKTEST');  
results.Properties.VariableNames
```

```
ans =
```

```
1x4 cell array
```

```
    {'SALARY'}    {'PLAYER'}    {'SIGNED_DATE'}    {'TEAM'}
```

- 8 Close the database connection.

```
close(conn)
```

See Also

`execute` | `fetch` | `database` | `close`

Related Examples

- “Insert Data into Database Table” on page 5-61

External Websites

- SQL Tutorial

Call Stored Procedure That Returns Data

This example shows how to call a stored procedure that returns data using the `fetch` function. Use the JDBC interface to connect to a Microsoft SQL Server database, call a stored procedure, and return data. For this example, the Microsoft SQL Server database contains the stored procedure `getSupplierInfo`. This stored procedure returns the supplier information for suppliers of a given city. This code defines the procedure.

```
CREATE PROCEDURE dbo.getSupplierInfo
    (@cityName varchar(20))
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON

    SELECT * FROM dbo.suppliers WHERE City = @cityName
END
```

For Microsoft SQL Server, the statement `'SET NOCOUNT ON'` suppresses the results of `INSERT`, `UPDATE`, or any non-`SELECT` statements that might be before the final `SELECT` query, so you can import the results of the `SELECT` query.

Use the `fetch` function when the stored procedure returns one or more result sets. For procedures that return output parameters, use `runstoredprocedure`.

Create Database Connection

Using the JDBC interface, connect to the Microsoft SQL Server database called `'test_db'` with a user name and password using port number 1234. This example assumes that your database server is on the machine `servername`.

```
conn = database('test_db','username','pwd', ...
    'Vendor','Microsoft SQL Server', ...
    'Server','servername','PortNumber',1234);
```

Call Stored Procedure

Call the stored procedure, `getSupplierInfo`, and display the supplier information for New York city using the `fetch` function and the database connection. `results` contains the supplier information.

```
sqlquery = '{call getSupplierInfo('New York')}';
results = fetch(conn,sqlquery)
```

ans =

3x5 table

SupplierNumber	SupplierName	City	Country	FaxNumber
1001	'Wonder Products'	'New York'	'United States'	'212 435 1617'
1006	'ACME Toy Company'	'New York'	'United States'	'212 435 1618'
1012	'Aunt Jemimas'	'New York'	'USA'	'14678923104'

Close Database Connection

`close(conn)`

See Also

`fetch` | `runstoredprocedure` | `database`

External Websites

- [SQL Tutorial](#)

Run Custom Database Function

This example shows how to run a custom database function on Microsoft SQL Server.

Consider a database function `get_prodCount` that retrieves row counts in the table `productTable`. The table `productTable` contains 30 rows where each row represents a product. This code defines this database function and assumes a schema name `dbo`.

```
CREATE FUNCTION dbo.get_prodCount()
RETURNS int
AS
BEGIN
    DECLARE @PROD_COUNT int
    SELECT @PROD_COUNT = count(*) FROM productTable
    RETURN(@PROD_COUNT)
END
GO
```

Create Database Connection

Connect to Microsoft SQL Server using an ODBC driver. For example, this code assumes that you are connecting to a data source named `MS SQL Server` with user name `username` and password `pwd`.

```
conn = database('MS SQL Server', 'username', 'pwd');
```

Execute Custom Function

Construct an SQL query `sqlquery` that executes the custom function code. Execute the custom function and import the results by using the `fetch` function.

```
sqlquery = 'SELECT dbo.get_prodCount() as num_products';
results = fetch(conn, sqlquery);
```

Display the results. The custom function `get_prodCount` returns the product count 30.

```
results
results =
  table
  num_products
  -----
      30
```

Close Database Connection

```
close(conn)
```

See Also

[fetch](#) | [database](#) | [close](#)

External Websites

- [SQL Tutorial](#)

Data Import Memory Management

To import data with simple queries, you can use the **Database Explorer** app. For more complex queries and managing memory issues, use the command line to import data into the MATLAB workspace. To understand the differences between these two approaches, see “Data Import Using Database Explorer App or Command Line” on page 2-125.

Database Toolbox provides various ways to import data into the MATLAB workspace from a database.

sqlread Function

If you are not familiar with writing SQL queries, you can import data using the `sqlread` function. This function needs only a database connection and the database table name to import data. Furthermore, the `sqlread` function does not require you to set database preferences.

select Function

For memory savings, you can import and access data using the `select` function. With this function, you save memory by importing data using data types specified in a database. The table definitions in a database specify the data type for each column. The `select` function maps the data type in the database to a corresponding MATLAB data type for each variable during data import. Instead of importing every numeric value as a `double` in MATLAB, the `select` function allows the import of different integer data types. You no longer need to convert the data type of a numeric value to a specific numeric type after data import. The MATLAB memory size used by integer or unsigned integer data types is less than double precision. Therefore, the `select` function saves memory.

This table shows the numeric data types in a database and their MATLAB equivalents when using the `select` function.

Database Data Type	MATLAB Data Type
SIGNED TINYINT	int8
UNSIGNED TINYINT	uint8
SIGNED SMALLINT	int16
UNSIGNED SMALLINT	uint16
SIGNED INT	int32
UNSIGNED INT	uint32
SIGNED BIGINT	int64
UNSIGNED BIGINT	uint64
REAL	single
FLOAT	single
DOUBLE	double
DECIMAL	double
NUMERIC	double
Boolean	logical
Date, time, or text	char

For example, create a table `Patients` with this database table definition:

```
CREATE TABLE Patients(  
    LastName VARCHAR(50),  
    Gender VARCHAR(10),  
    Age TINYINT,  
    Location VARCHAR(300),  
    Height SMALLINT,  
    Weight SMALLINT,  
    Smoker BIT,  
    Systolic FLOAT,  
    Diastolic NUMERIC,  
    SelfAssessedHealthStatus VARCHAR(20))
```

These table columns have numeric data types in the database:

- Age
- Height
- Weight
- Systolic
- Diastolic

The `fetch` function imports the columns of numeric data with double precision by default. However, the `select` function imports the columns into their matching integer data type. When you import using the `select` function, the corresponding MATLAB data types for these columns are:

- `uint8`
- `uint16`
- `uint16`
- `single`
- `double`

The `fetch` function imports the `Smoker` column as a `double` in MATLAB. However, the `select` function imports the `Smoker` column as a `logical` variable.

To see data types after data import, use the `select` function with the `metadata` output argument.

Define Import Strategy Using `SQLImportOptions` Object

You can customize the import options for importing data from a database into the MATLAB workspace by using the `SQLImportOptions` object with the `fetch` function. The `select` function specifies the MATLAB data type by default. However, with the `SQLImportOptions` object, you can define the import strategy for specific database columns and specify the MATLAB data type for the corresponding imported data.

Also, you can specify `categorical`, `datetime`, and integer data types for imported data using the `SQLImportOptions` object. The MATLAB memory size used to store these data types is less than the memory size used for alternative data types, such as `string` or `double`.

See Also

`fetch` | `executeSQLScript` | `select` | `sqlread`

More About

- “Data Import Using Database Explorer App or Command Line” on page 2-125
- “Import Data from Database Table Using sqlread Function” on page 5-58
- “Working with Large Data Sets” on page 2-127

Import Large Data Using DatabaseDatastore Object

This example shows how to create a `DatabaseDatastore` object for accessing collections of data stored in a relational database. After creating a `DatabaseDatastore` object, you can preview data, read data in chunks, and read every record in the data set.

To analyze large data, you can run algorithms on large data sets using a tall array.

Alternatively, you can write a MapReduce algorithm that defines the chunking and reduction of the data.

This example uses a preconfigured JDBC data source to create the database connection. For more information, see the `configureJDBCDataSource` function.

Create DatabaseDatastore Object

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
username = "";
password = "";
conn = database(datasource,username,password);
```

Create a `DatabaseDatastore` object using the database connection and an SQL query. This SQL query retrieves all data from the `airlinesmall` table.

```
sqlquery = 'select * from airlinesmall';
dbds = databaseDatastore(conn,sqlquery);
```

Preview Data in DatabaseDatastore Object

Preview the first eight records in the data set returned by executing the SQL query.

```
preview(dbds)
```

```
ans =
```

```
8×29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1990	9	11	2	1810	1812	1939	1930
1990	10	27	6	1353	1355	1634	1640
1990	10	23	2	1057	1055	1205	1155
1990	10	8	1	1515	1440	1609	1535
1990	10	19	5	1130	1120	1203	1154
1990	10	12	5	1755	1733	1858	1820
2001	11	22	4	1345	1355	1530	1549
2001	11	26	1	2105	2110	2209	2237

Read Data in DatabaseDatastore Object

Read the first 10 records.

```
dbds.ReadSize = 10;
```

```
read(dbds)
```

```
ans =
```

```
10×29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	30	5	1329	1329	1434	1436
1987	11	7	6	1316	1315	1713	1647
1987	11	28	6	815	815	1015	1015
1987	11	2	1	700	700	800	800
1987	11	14	6	840	840	1127	1120
1987	11	1	7	1625	1625	1823	1758
1987	11	26	4	1314	1315	1538	1542
1987	10	28	3	1140	1140	1212	1215
1987	10	9	5	1155	1155	1250	1300
1987	10	22	4	715	715	807	803

Read the DatabaseDatastore object two more times by using counter n. Read 10 records at a time.

```
n = 0;
```

```
while(hasdata(dbds) && n~=2)
    read(dbds)
    n = n+1;
end
```

```
ans =
```

```
10×29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	16	5	1553	1555	1641	1640
1987	10	30	5	1821	1815	1956	1955
1987	10	12	1	1300	1300	1529	1528
1987	10	7	3	810	810	904	900
1987	10	19	1	733	735	827	831
1987	10	15	4	828	830	916	921
1987	10	4	7	1750	1735	1837	1816
1987	10	16	5	959	1000	1212	1215
1987	10	17	6	2020	2020	2100	2057
1987	10	6	2	1132	1135	1426	1419

```
ans =
```

10×29 table

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	24	6	944	945	1211	1213
1987	10	18	7	833	835	1003	1011
1987	10	26	1	2356	2355	730	721
1987	10	29	4	1056	1055	1208	1215
1987	10	1	4	2304	2255	2340	2329
1987	10	3	6	1040	1040	1125	1120
1987	10	23	5	1855	1855	2158	2205
1987	10	30	5	1055	1055	1302	1315
1987	10	28	3	NaN	1850	NaN	2050
1987	10	26	1	1600	1600	1649	1640

Reset DatabaseDatastore Object

Reset the DatabaseDatastore object to its original state, where no data has been read from it. Resetting allows you to reread from the same DatabaseDatastore object.

```
reset(dbds)
```

Read Every Record in DatabaseDatastore Object

Read every record in the DatabaseDatastore object in increments of 50,000 records at a time.

```
dbds.ReadSize = 50000;
data = readall(dbds);
```

Display the first three records of the full data set.

```
data(1:3, :)
```

```
ans =
```

3×29 table

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	28	3	1140	1140	1212	1215
1987	10	9	5	1155	1155	1250	1300
1987	10	22	4	715	715	807	803

Close DatabaseDatastore Object and Database Connection

```
close(dbds)
```

See Also

[database](#) | [databaseDatastore](#) | [close](#) | [hasdata](#) | [preview](#) | [read](#) | [readall](#) | [reset](#)

Related Examples

- “Analyze Large Data in Database Using Tall Arrays” on page 5-30
- “Analyze Large Data in Database Using MapReduce” on page 5-27
- Building Effective Algorithms with MapReduce

External Websites

- [SQL Tutorial](#)

Analyze Large Data in Database Using MapReduce

This example determines the mean arrival delay of a large set of flight data that is stored in a database. You can access large data sets using a `databaseDatastore` object with Database Toolbox™. After creating a `DatabaseDatastore` object, you can write a MapReduce algorithm that defines the chunking and reduction of the data. Alternatively, you can use a tall array to run algorithms on large data sets.

The `DatabaseDatastore` object does not support using a parallel pool with Parallel Computing Toolbox™ installed. To analyze data using tall arrays or run MapReduce algorithms, set the global execution environment to be the local MATLAB® session.

This example uses a preconfigured JDBC data source to create the database connection. For more information, see the `configureJDBCDataSource` function.

Create DatabaseDatastore Object

Set the global execution environment to be the local MATLAB® session.

```
mapreducer(0);
```

The file `airlinesmall.csv` contains a large set of flight data. Load this file into the Microsoft® SQL Server® database table `airlinesmall`. This table contains 123,523 records.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
username = "";
password = "";
conn = database(datasource,username,password);
```

Create a `DatabaseDatastore` object using the database connection and an SQL query. This SQL query retrieves arrival-delay data from the `airlinesmall` table.

```
sqlquery = 'select ArrDelay from airlinesmall';
```

```
dbds = databaseDatastore(conn,sqlquery);
```

Define Mapper and Reducer Functions

To process large data sets in chunks, you can write your own mapper function. For each chunk in this example, use `meanArrivalDelayMapper.m` to:

- Read arrival-delay data from the `DatabaseDatastore` object.
- Determine the number of delays and the total delay in the chunk.
- Store both values in `KeyValueDatastore`.

The `meanArrivalDelayMapper.m` file contains this code.

```
function meanArrivalDelayMapper (data, info, intermKVStore)
% Mapper function for the MeanMapReduceExample.
```

```

% Copyright 2014 The MathWorks, Inc.

% Data is an n-by-1 table of the ArrDelay. Remove missing value first:
data(isnan(data.ArrDelay),:) = [];

% Record the partial counts and sums and the reducer will accumulate them.
partCountSum = [length(data.ArrDelay), sum(data.ArrDelay)];
add(intermKVStore, 'PartialCountSumDelay',partCountSum);

```

You also can write your own reducer function. In this example, use `meanArrivalDelayReducer.m` to read intermediate values for the number of delays and the total arrival delay. Then, determine the overall mean arrival delay. `mapreduce` calls this reducer function only once because the mapper function adds just one key to `KeyValueStore`. The `meanArrivalDelayReducer.m` file contains this code.

```

function meanArrivalDelayReducer(intermKey, intermValIter, outKVStore)
% Reducer function for the MeanMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

% intermKey is 'PartialCountSumDelay'
count = 0;
sum = 0;
while hasNext(intermValIter)
    countSum = getNext(intermValIter);
    count = count + countSum(1);
    sum = sum + countSum(2);
end

meanDelay = sum/count;

% The key-value pair added to outKVStore will become the output of mapreduce
add(outKVStore, 'MeanArrivalDelay',meanDelay);

```

Run MapReduce Using Mapper and Reducer Functions

To determine the mean arrival delay in the flight data, run `MapReduce` with the `DatabaseDatastore` object, mapper function, and reducer function.

```

outds = mapreduce(dbds,@meanArrivalDelayMapper,@meanArrivalDelayReducer);

```

```

*****
*      MAPREDUCE PROGRESS      *
*****
Map   0% Reduce   0%
Map  15% Reduce   0%
Map  30% Reduce   0%
Map  46% Reduce   0%
Map  61% Reduce   0%
Map  76% Reduce   0%
Map  92% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%

```

Display Output from MapReduce

Read the table from the output datastore using `readall`.

```
outtab = readall(outds)
```

```
outtab =
```

```
1×2 table
```

Key	Value
'MeanArrivalDelay'	[7.1201]

The table has only one row containing one key-value pair.

Display the mean arrival delay from the table.

```
meanArrDelay = outtab.Value{1}
```

```
meanArrDelay =
```

```
7.1201
```

Close DatabaseDatastore Object and Database Connection

```
close(dbds)
```

See Also

`setdbprefs` | `database` | `databaseDatastore` | `readall` | `close` | `mapreduce` | `TabularTextDatastore`

Related Examples

- “Analyze Large Data in Database Using Tall Arrays” on page 5-30
- “Compute Mean Value with MapReduce”
- “Import Large Data Using DatabaseDatastore Object” on page 5-23
- Building Effective Algorithms with MapReduce
- “Getting Started with MapReduce”

External Websites

- SQL Tutorial

Analyze Large Data in Database Using Tall Arrays

This example determines the minimum arrival delay of a large set of flight data that is stored in a database. You can access large data sets and create a tall array using a `DatabaseDatastore` object with Database Toolbox™. Once a tall array exists, you can visualize data in the tall array. Alternatively, you can write a MapReduce algorithm that defines the chunking and reduction of the data.

The `DatabaseDatastore` object does not support using a parallel pool with Parallel Computing Toolbox™ installed. To analyze data using tall arrays or run MapReduce algorithms, set the global execution environment to be the local MATLAB® session.

This example uses a preconfigured JDBC data source to create the database connection. For more information, see the `configureJDBCDataSource` function.

Create DatabaseDatastore Object

Set the global execution environment to be the local MATLAB® session.

```
mapreducer(0);
```

The file `airlinesmall.csv` contains the large set of flight data. Load this file into the Microsoft® SQL Server® database table `airlinesmall`. This table contains 123,523 records.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
username = "";
password = "";
conn = database(datasource,username,password);
```

Create a `DatabaseDatastore` object using the database connection and an SQL query. This SQL query retrieves arrival-delay data from the `airlinesmall` table. `databaseDatastore` executes the SQL query.

```
sqlquery = 'select ArrDelay from airlinesmall';
dbds = databaseDatastore(conn,sqlquery,'ReadSize',50000);
```

Find Minimum Arrival Delay Using Tall Array

Because the `DatabaseDatastore` object returns a table, create a tall table.

```
tt = tall(dbds);
```

Find the minimum arrival delay.

```
minArrDelay = min(tt.ArrDelay);
```

`minArrDelay` contains the unevaluated minimum arrival delay. To return the output value, use `gather`. For details, see “Deferred Evaluation of Tall Arrays”.

```
minArrDelayValue = gather(minArrDelay)
```



```
Evaluating tall expression using the Local MATLAB Session:  
- Pass 1 of 1: Completed in 1.6 sec  
Evaluation completed in 1.9 sec
```

```
minArrDelayValue =  
    -64
```

In addition to determining a minimum, tall arrays support many other functions. For details, see “Supporting Functions”.

Close DatabaseDatastore Object and Database Connection

```
close(dbds)
```

See Also

databaseDatastore | database | gather | min | histogram | mapreducer | tall

Related Examples

- “Visualization of Tall Arrays”
- “Analyze Large Data in Database Using MapReduce” on page 5-27

More About

- “Tall Arrays for Out-of-Memory Data”

External Websites

- SQL Tutorial

Import Data Using MATLAB Interface to SQLite

This example shows how to move data between MATLAB® and the MATLAB interface to SQLite. Suppose that you have product data that you want to import into MATLAB. You can load this data quickly into an SQLite database file. You do not need to install a database or driver. For details about the MATLAB interface to SQLite, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Create SQLite Connection

Create an SQLite connection `conn` to a new SQLite database file `tutorial.db`. Specify the file name in the current working folder.

```
dbfile = fullfile(pwd,"tutorial.db");
conn = sqlite(dbfile,"create");
```

Create Tables in SQLite Database File

Create the tables `inventoryTable`, `suppliers`, `salesVolume`, and `productTable` using `execute`. Clear the MATLAB workspace variables.

```
createInventoryTable = strcat("CREATE TABLE inventoryTable ", ...
    "(productNumber NUMERIC, Quantity NUMERIC, ", ...
    "Price NUMERIC, inventoryDate VARCHAR)");
execute(conn,createInventoryTable)
```

```
createSuppliers = strcat("CREATE TABLE suppliers ", ...
    "(SupplierNumber NUMERIC, SupplierName VARCHAR(50), ", ...
    "City VARCHAR(20), Country VARCHAR(20), ", ...
    "FaxNumber VARCHAR(20))");
execute(conn,createSuppliers)
```

```
createSalesVolume = strcat("CREATE TABLE salesVolume ", ...
    "(StockNumber NUMERIC, January NUMERIC, ", ...
    "February NUMERIC, March NUMERIC, April NUMERIC, ", ...
    "May NUMERIC, June NUMERIC, July NUMERIC, ", ...
    "August NUMERIC, September NUMERIC, October NUMERIC, ", ...
    "November NUMERIC, December NUMERIC)");
execute(conn,createSalesVolume)
```

```
createProductTable = strcat("CREATE TABLE productTable ", ...
    "(productNumber NUMERIC, stockNumber NUMERIC, ", ...
    "supplierNumber NUMERIC, unitCost NUMERIC, ", ...
    "productDescription VARCHAR(20))");
execute(conn,createProductTable)
```

```
clear createInventoryTable createSuppliers createSalesVolume ...
    createProductTable
```

`tutorial.db` contains four empty tables.

Load Data into SQLite Database File

Load the MAT-file named `sqliteworkflowdata.mat`. The variables `CinvTable`, `Csuppliers`, `CsalesVol`, and `CprodTable` contain data for export. Export data into the tables in `tutorial.db` using `sqlwrite`. Clear the MATLAB workspace variables.

```

load('sqliteworkflowdata.mat')

tablename = "inventoryTable";
varnames = ["productNumber" "Quantity" "Price" "inventoryDate"];
data = cell2table(CinvTable); % convert data from cell array to table
data.Properties.VariableNames = varnames; % set variable names
sqlwrite(conn,tablename,data)

tablename = "suppliers";
varnames = ["SupplierNumber" "SupplierName" "City" "Country" "FaxNumber"];
data = cell2table(Csuppliers); % convert data from cell array to table
data.Properties.VariableNames = varnames; % set variable names
sqlwrite(conn,tablename,data)

tablename = "salesVolume";
varnames = ["StockNumber" "January" "February" "March" "April" "May" "June" ...
            "July" "August" "September" "October" "November" "December"];
data = cell2table(CsalesVol); % convert data from cell array to table
data.Properties.VariableNames = varnames; % set variable names
sqlwrite(conn,tablename,data)

tablename = "productTable";
varnames = ["productNumber" "stockNumber" "supplierNumber" "unitCost" ...
            "productDescription"];
data = cell2table(CprodTable); % convert data from cell array to table
data.Properties.VariableNames = varnames; % set variable names
sqlwrite(conn,tablename,data)

clear CinvTable Csuppliers CsalesVol CprodTable

Close the SQLite connection. Clear the MATLAB workspace variable.
close(conn)

clear conn

Create a read-only SQLite connection to tutorial.db.
conn = sqlite("tutorial.db","readonly");

```

Import Data into MATLAB

Import the product data into the MATLAB workspace using `fetch`. Variables `inventoryTable_data`, `suppliers_data`, `salesVolume_data`, and `productTable_data` contain data from the tables `inventoryTable`, `suppliers`, `salesVolume`, and `productTable`.

```

inventoryTable_data = fetch(conn,"SELECT * FROM inventoryTable");

suppliers_data = fetch(conn,"SELECT * FROM suppliers");

salesVolume_data = fetch(conn,"SELECT * FROM salesVolume");

productTable_data = fetch(conn,"SELECT * FROM productTable");

```

Display the first three rows of data in each table.

```
head(inventoryTable_data,3)
```

<u>productNumber</u>	<u>Quantity</u>	<u>Price</u>	<u>inventoryDate</u>
----------------------	-----------------	--------------	----------------------

```

1          1700      14.5    "9/23/2014 9:38:34 AM"
2          1200       9.3    "7/8/2014 10:50:45 PM"
3           356     17.2    "5/14/2014 7:14:28 AM"

```

```
head(suppliers_data,3)
```

SupplierNumber	SupplierName	City	Country	FaxNumber
1001	"Wonder Products"	"New York"	"United States"	"212 435 1617"
1002	"Terrific Toys"	"London"	"United Kingdom"	"44 456 9345"
1003	"Wacky Widgets"	"Adelaide"	"Australia"	"618 8490 2211"

```
head(salesVolume_data,3)
```

StockNumber	January	February	March	April	May	June	July	August	September
125970	1400	1100	981	882	794	752	654	773	800
212569	2400	1721	1414	1191	983	825	731	653	700
389123	1800	1200	890	670	550	450	400	410	400

```
head(productTable_data,3)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	125970	1003	13	"Victorian Doll"
8	212569	1001	5	"Train Set"
7	389123	1007	16	"Engine Kit"

Close SQLite Connection

```
close(conn)
```

Clear the MATLAB workspace variable.

```
clear conn
```

See Also

Objects

sqlite

Functions

execute | fetch | sqlwrite | close

Related Examples

- "Interact with Data in SQLite Database Using MATLAB Interface to SQLite" on page 2-5
- "Insert Data into SQLite Database Table" on page 5-36
- "Create Table and Add Column in SQLite Database" on page 5-38

- “Delete Data from SQLite Database” on page 5-39
- “Deploy MATLAB Interface to SQLite Database Application with MATLAB Compiler” on page 5-43

External Websites

- [SQL Tutorial](#)

Insert Data into SQLite Database Table

This example shows how to import data from an SQLite database into MATLAB® using the MATLAB interface to SQLite, perform calculations on the data, and export the results to a database table.

The example assumes that you connect to an SQLite database that contains tables named `salesVolume` and `yearlySales`. The `salesVolume` table contains the column names for each month. The `yearlySales` table contains the column names `Month` and `SalesTotal`.

Create SQLite Connection

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";  
conn = sqlite(dbfile);
```

Calculate Sum of Sales Volume for One Month

Import sales volume data for the month of March using the database connection. The `salesVolume` database table contains sales volume data.

```
tablename = "salesVolume";  
data = sqlread(conn,tablename);
```

Display the first three rows of sales volume data. The fourth variable contains the data for the month of March.

```
head(data(:,4),3)
```

```
    March  
    _____  
    981  
    1414  
    890
```

Calculate the sum of the March sales. Assign the result to the MATLAB workspace variable `total` and display the result.

```
total = sum(data.March)  
  
total = 27609
```

Insert Total Sales for One Month into Database

Retrieve the name of the month from the sales volume data.

```
month = data.Properties.VariableNames(4);
```

Define the names of the columns for the data to insert as a string array.

```
colnames = ["Month" "SalesTotal"];
```

Create a MATLAB table that stores the data to export.

```
results = table(month,total,'VariableNames',colnames);
```

Determine the status of the `AutoCommit` database flag. This status shows whether or not the insert action can be undone.

```
conn.AutoCommit
```

```
ans =  
'on'
```

The `AutoCommit` flag is set to `on`. The database commits the exported data automatically to the database, and this action cannot be undone.

Insert the sum of sales for the month of March into the `yearlySales` table.

```
tablename = "yearlySales";  
sqlwrite(conn,tablename,results)
```

Import the data from the `yearlySales` table. This data contains the calculated result.

```
data = sqlread(conn,tablename)
```

```
data=1x2 table  
      Month      SalesTotal  
      _____  _____  
      "March"      27609
```

Close Database Connection

```
close(conn)
```

See Also

Objects

sqlite

Functions

sqlread | sqlwrite | close

Related Examples

- “Import Data Using MATLAB Interface to SQLite” on page 5-32
- “Create Table and Add Column in SQLite Database” on page 5-38
- “Delete Data from SQLite Database” on page 5-39
- “Roll Back Data in SQLite Database” on page 5-41

Create Table and Add Column in SQLite Database

This example shows how to connect to an SQLite database and manage the database structure using the MATLAB® interface to SQLite. You can manage the database structure using the `execute` function.

Create SQLite Connection

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";  
conn = sqlite(dbfile);
```

Create Database Table

Use the SQL `CREATE` statement to create the database table `Person`.

```
sqlquery = strcat("CREATE TABLE Person(lastname VARCHAR(250), ", ...  
    "firstname VARCHAR(250), address VARCHAR(300), age INT)");
```

Create the table in the database using the database connection.

```
execute(conn,sqlquery)
```

Add Database Column

Use the SQL `ALTER` statement to add the database column `city` to the table `Person`.

```
sqlquery = "ALTER TABLE Person ADD city VARCHAR(30)";  
execute(conn,sqlquery)
```

Close Database Connection

```
close(conn)
```

See Also

Objects

`sqlite`

Functions

`execute` | `close`

Related Examples

- “Import Data Using MATLAB Interface to SQLite” on page 5-32
- “Delete Data from SQLite Database” on page 5-39
- “Roll Back Data in SQLite Database” on page 5-41

External Websites

- SQL Tutorial

Delete Data from SQLite Database

This example shows how to delete data from an SQLite database using the MATLAB® interface to SQLite. Create the SQL statement using deletion SQL syntax. Execute the delete operation on your database using the `execute` function with the SQL statement.

Create SQLite Connection

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

The SQL query `sqlquery` selects all rows of data in the table `inventoryTable`. Execute this SQL query using the database connection. Import the data from the executed query using the `fetch` function, and display the last few rows.

```
sqlquery = "SELECT * FROM inventoryTable";
data = fetch(conn,sqlquery);
tail(data,3)
```

productNumber	Quantity	Price	inventoryDate
11	567	11.2	"9/11/2012 12:30:24 AM"
12	1278	22.3	"10/29/2010 6:17:47 PM"
13	1700	16.8	"5/24/2009 10:58:59 AM"

Delete Specific Record

Delete the record for the product number 13 from the table `inventoryTable`. Specify the product number using the `WHERE` clause in the SQL statement `sqlquery`.

```
sqlquery = "DELETE FROM inventoryTable WHERE productnumber = 13";
execute(conn,sqlquery)
```

Display the data in the table `inventorytable` after the deletion. The record with product number 13 is missing.

```
sqlquery = "SELECT * FROM inventoryTable";
data = fetch(conn,sqlquery);
tail(data,3)
```

productNumber	Quantity	Price	inventoryDate
10	723	24.3	"3/14/2012 1:13:09 PM"
11	567	11.2	"9/11/2012 12:30:24 AM"
12	1278	22.3	"10/29/2010 6:17:47 PM"

Insert the record back in to maintain the dataset.

```
data = table(13,1700,16.8,"5/24/2009 10:58:59 AM", ...
    'VariableNames',{'productNumber' ...
    'Quantity' 'Price' 'inventoryDate'});
sqlwrite(conn,"inventoryTable",data)
```

Close Database Connection

`close(conn)`

See Also

Objects

`sqlite`

Functions

`execute` | `fetch` | `close`

Related Examples

- “Import Data Using MATLAB Interface to SQLite” on page 5-32
- “Create Table and Add Column in SQLite Database” on page 5-38
- “Roll Back Data in SQLite Database” on page 5-41

External Websites

- [SQL Tutorial](#)

Roll Back Data in SQLite Database

This example shows how to connect to an SQLite database, insert a row into an existing database table, and then roll back the insertion using the MATLAB® interface to SQLite. The database contains the table `productTable`.

Create SQLite Connection

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Append Data to Existing Database Table

Set the `AutoCommit` property of the connection object to `off`. Any updates you make after turning off this flag do not commit to the database automatically.

```
conn.AutoCommit = "off";
```

To view the existing database table `productTable` before appending data, import its contents into MATLAB and display the last few rows.

```
tablename = "productTable";
rows = sqlread(conn,tablename);
tail(rows,3)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	470816	1012	16	"Pancakes"
14	510099	1011	19	"Shawl"
15	899752	1011	20	"Snacks"

Create a MATLAB table that contains the data for one product.

```
data = table(30,500000,1000,25,"Rubik's Cube", ...
    'VariableNames',["productnumber" "stocknumber" ...
    "suppliernumber" "unitcost" "productdescription"]);
```

Append the product data into the database table `productTable`.

```
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB again and display the last few rows. The results contain a new row for the inserted product.

```
rows = sqlread(conn,tablename);
tail(rows,4)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	470816	1012	16	"Pancakes"
14	510099	1011	19	"Shawl"
30	500000	1000	25	"Rubik's Cube"

15	899752	1011	20	"Snacks"
30	500000	1000	25	"Rubik's Cube"

Roll Back Data

Roll back the inserted row.

```
rollback(conn)
```

Import the contents of the database table into MATLAB again and display the last few rows. The results no longer contain the inserted row.

```
rows = sqlread(conn,tablename);  
tail(rows,3)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	470816	1012	16	"Pancakes"
14	510099	1011	19	"Shawl"
15	899752	1011	20	"Snacks"

Close Database Connection

```
close(conn)
```

See Also

Objects

sqlite

Functions

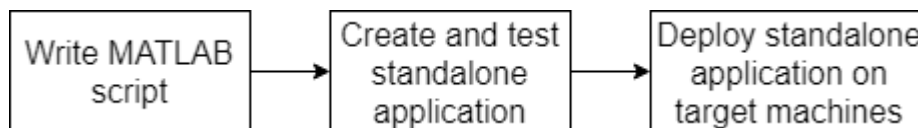
sqlread | sqlwrite | rollback | close

Related Examples

- "Import Data Using MATLAB Interface to SQLite" on page 5-32
- "Create Table and Add Column in SQLite Database" on page 5-38
- "Delete Data from SQLite Database" on page 5-39

Deploy MATLAB Interface to SQLite Database Application with MATLAB Compiler

This example shows how to write a script to analyze data stored in an SQLite database using the MATLAB® interface to SQLite, and then deploy the script as a standalone application. For this example, you write code that connects to a database, imports data from the database into MATLAB, analyzes the data, and closes the database connection. Then, you deploy the code by compiling it as a standalone application using the Application Compiler (MATLAB Compiler) app, and by running the application on other machines.



The example uses the MATLAB interface to SQLite to create the database connection. Overall, the example follows the steps described in “Create Standalone Application from MATLAB Function” (MATLAB Compiler) and updates the steps for a standalone database application.

Create Function in MATLAB

Create a MATLAB script named `importSQLiteInterface.m` and save it in a file location of your choice. The script contains the `importSQLiteInterface` function, which returns the maximum product number from the data in the `productTable` database table. The function connects to the SQLite database and imports all data from `productTable`. Then, the function calculates the maximum product number.

type `importSQLiteInterface.m`

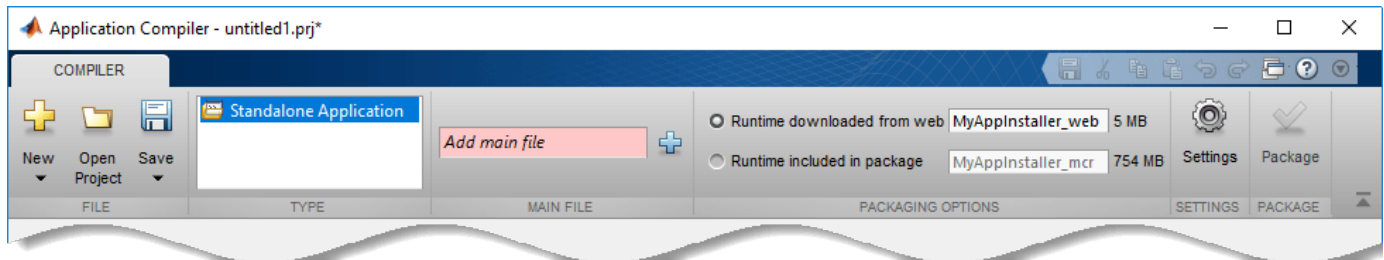
```

function maxProdNum = importSQLiteInterface
% IMPORTSQLITEINTERFACE The importSQLiteInterface function connects to an
% SQLite database using the MATLAB(R) interface to SQLite, imports data
% from the database into MATLAB, performs a simple data analysis, and
% closes the database connection. The database contains a table named
% |productTable|.


%%
% Create the SQLite connection |conn| to the existing SQLite database file
% |tutorial.db|. The SQLite connection is an |sqlite| object.
dbfile = "tutorial.db";
conn = sqlite(dbfile);
%%
% Import data from the |productTable| database table.
tablename = "productTable";
data = sqlread(conn,tablename);
%%
% Determine the highest product number among products.
prodNums = data.productNumber;
maxProdNum = max(prodNums);
%%
% Close the database connection.
close(conn)
end
  
```

Create Standalone Application Using Application Compiler App

On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow to open the apps gallery. Under **Application Deployment**, click **Application Compiler**.



In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.

- 1 In the **Main File** section of the toolstrip, click the plus sign button  to add a main file.
- 2 In the **Add Files** dialog box, browse to the file location that contains your saved script. Select `importSQLiteInterface.m` and click **Open**. The Application Compiler app adds the `importSQLiteInterface` function to the list of main files.

Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:

- **Runtime downloaded from web** — Generates an installer that downloads MATLAB Runtime and installs it along with the deployed MATLAB application.
- **Runtime included in package** — Generates an installer that includes the MATLAB Runtime installer.

Customize the packaged application and its appearance by specifying the following options:

The screenshot displays the MATLAB Compiler application configuration window. The 'Application information' section is active, showing a form with the following fields: 'Application Name' (with a version of 1.0), 'Author Name', 'Email', 'Company', 'Summary', and 'Description'. A 'Set as default contact' link is located next to the 'Company' field. To the right, a splash screen preview is shown with a 'Select custom splash screen' button. Below the main form, there are four expandable sections: 'Additional installer options', 'Files required for your application to run', 'Files installed for your end user', and 'Additional runtime settings'. Each of the latter three sections has a '+' button in the bottom right corner.

- **Application information** — Editable information about the deployed application. You can also customize the appearance of the standalone application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata.
- **Additional installer options** — Options for editing the default installation path for the generated installer and selecting a custom logo.
- **Files required for your application to run** — Additional files required by the generated application to run. The software includes these files in the generated application installer.
- **Files installed for your end user** — Files that are installed with your application. These files include the generated `readme.txt` file and the generated executable for the target platform.
- **Additional runtime settings** — Platform-specific options for controlling the generated executable.

For details about these options, see “Customize an Application” (MATLAB Compiler).

To generate the packaged application, click **Package** in the **Package** section on the toolbar. In the Save Project dialog box, specify the location in which to save the project.

In the **Package** dialog box, verify that the option **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output:

- `for_redistribution` — Folder containing the file that installs the application and MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc` (such as binary, JAR, header, and source files for a specific target). Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler.

Install and Run Standalone Application

To install the standalone application, double-click the `MyAppInstaller_web` executable in the `for_redistribution` folder.

If you want to connect to the Internet using a proxy server, click **Connection Settings**. Enter the proxy server settings in the provided dialog box. Click **OK**.

To complete the installation, follow the instructions in the installation wizard.

To run your standalone application:

- 1 Open a terminal window.
- 2 Navigate to the folder in which you installed the application.
- 3 Run the application.

Test Standalone Application on Target Machine

Choose one target machine to test the MATLAB generated standalone application.

Copy the files in the `for_testing` folder to the target machine.

To test your standalone application:

- 1 Open a terminal window.
- 2 Navigate to the `for_testing` folder.
- 3 Run the application.

Deploy Standalone Application on Target Machines

Copy the `for_redistribution_files_only` folder to a file location on all target machines where MATLAB or MATLAB Runtime is installed.

Run the MATLAB generated standalone application on all target machines by using the executable in the `for_redistribution_files_only` folder.

See Also

Objects

`sqlite`

Functions

`sqlread` | `close`

Related Examples

- “Create Functions in Files”
- “Create Standalone Application from MATLAB Function” (MATLAB Compiler)
- “Customize an Application” (MATLAB Compiler)
- “Import Data Using MATLAB Interface to SQLite” on page 5-32

External Websites

- [SQL Tutorial](#)

Retrieve Image Data Types

This example shows how to retrieve images from a Microsoft Access database. To run this example, define the function `parsebinary` using this code.

```
function [x,map] = parsebinary(o,f)
%PARSEBINARY Write binary object to disk and display if image.
% [X,MAP] = PARSEBINARY(O,F) writes the binary object in O to disk
% in the format specified by F. If the object is an image,
% display the image. This file was released for demonstration
% purposes only. A Microsoft(R) Access(TM) database contains image data.
% Use an ODBC driver to read the data. This function writes
% any temporary files to the current working directory.
%
% Valid file formats are:
%
% BMP      Bitmap
% DOC      Microsoft(R) Word document
% GIF      GIF file
% PPT      Microsoft(R) Powerpoint(R) file
% TIF      TIF file
% XLS      Microsoft(R) Excel(R) spreadsheet
% PNG      Portable Network Graphics

% Transform object into vector of data
v = java.util.Vector;
v.addElement(o);
bdata = v.elementAt(0);

% Open file to write data to disk
fid = fopen(['testfile.' lower(f)], 'wb');

% n specifies the end point of data written to disk
n = length(bdata);

% File type determines how many bytes of header data that
% the ODBC driver prepended to the data.

switch lower(f)

    case 'bmp'
        m = 79;

    case 'doc'
        m = 86;

    case 'gif'
        m = 5722;

    case 'png'

        m = 182;
        n = length(bdata)-285;

    case 'ppt'
        m = 94;

    case 'tif'
        m = 6472;

    case 'xls'
        m = 83;

    otherwise
        error(message('database:parsebinary:unknownFormat'))

end

% Write data to disk
fwrite(fid,bdata(m:n),'int8');
fclose(fid);

% Display if image
switch lower(f)

    case {'bmp','tif','gif','png'}

        [x,map] = imread(['testfile.' lower(f)]);
        imagesc(x)
```

```

colormap(map)
case {'doc','xls','ppt'}
% Microsoft(R) Office formats
% Insert path to Microsoft(R) Word or Microsoft(R) Excel(R)
% executable here to run from MATLAB(R) prompt.
% For example:
% !d:\msoffice\winword testfile.doc
end

```

- 1 Connect to the Microsoft Access data source using the ODBC driver. The database contains the table Invoice.

```
conn = database('datasource','','');
```

- 2 Import the InvoiceNumber and Receipt columns of data from Invoice.

```
sqlquery = 'SELECT InvoiceNumber,Receipt FROM Invoice';
results = fetch(conn,sqlquery);
```

- 3 View the imported data.

```
results
```

```
ans =
```

```
10x2 table
```

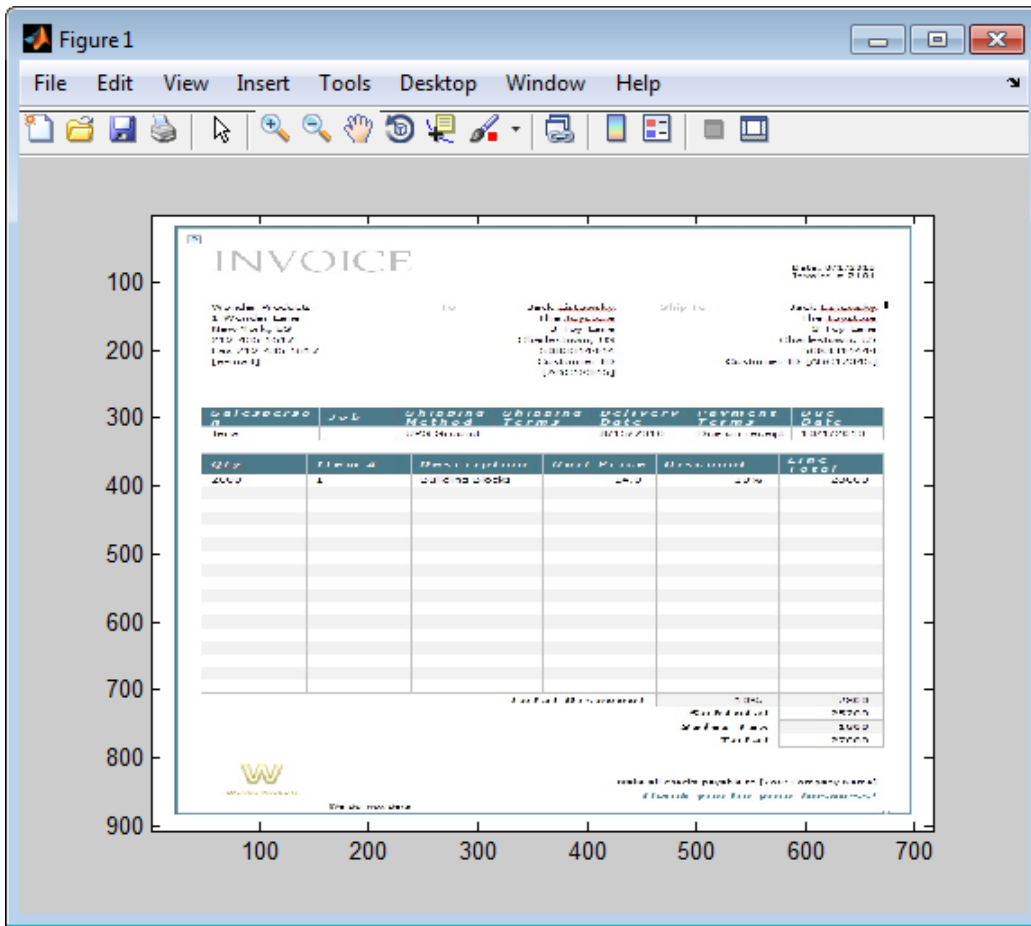
InvoiceNumber	Receipt
2101	[1948410x1 uint8]
3546	[2059994x1 uint8]
33116	[487034x1 uint8]
34155	[2059994x1 uint8]
34267	[2454554x1 uint8]
37197	[1926362x1 uint8]
37281	[2403674x1 uint8]
41011	[1920474x1 uint8]
61178	[2378330x1 uint8]
62145	[492314x1 uint8]

- 4 Assign the image element you want to the variable receipt.

```
receipt = results.Receipt{1};
```

- 5 Run the parsebinary function. The function writes retrieved data to a file, strips ODBC header information from it, and displays receipt as a bitmap image in a figure window. Ensure that your current folder is writable so that the parsebinary function can write the output data.

```
cd 'I:\MATLABFiles\myfiles'
parsebinary(receipt,'BMP');
```



See Also

database | fetch

Related Examples

- “Import Data from Database Table Using sqlread Function” on page 5-58

External Websites

- SQL Tutorial

Import Boolean Data from Database

Import Boolean data from a database table into the MATLAB® workspace. MATLAB® imports Boolean data from databases into the MATLAB® workspace as data type `logical`. This data has values of `true` or `false`. You can store Boolean data in a table, structure, or cell array. Perform a simple data analysis on the imported data.

The code assumes that you have a database table `Invoice` stored in a Microsoft® SQL Server® database. Here, connect to a Microsoft® SQL Server® Version 11.00.2100 database using the Microsoft® SQL Server® Driver 11.00.5058.

Create Database Connection

Create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Import Boolean Data

Select the paid data from the `Invoice` table using an SQL `SELECT` statement. The database stores paid data as a Boolean to specify whether or not an invoice has been paid. Import and display the data using the `select` function.

```
selectquery = 'SELECT Paid FROM Invoice';
data = select(conn,selectquery)
```

```
data =
    10×1 table
    Paid
    _____
    false
    true
    true
    false
    true
    true
    false
    true
    false
    true
```

Database Toolbox™ imports the data into the workspace variable `data`. The MATLAB® table `data` contains `Paid` as a `logical` variable.

Perform Data Analysis

Count the number of unpaid invoices.

```
unpaid = data.Paid == false;  
sum(unpaid)
```

```
ans =  
     4
```

Close Database Connection

```
close(conn)
```

See Also

database | select | close

Related Examples

- “Import Data from Database Table Using sqlread Function” on page 5-58
- “Insert Data into Database Table” on page 5-61

External Websites

- [SQL Tutorial](#)

Append Data to Existing Database Table Using Insert Functionality

To append data to an existing database table, you can use the `sqlwrite` function. The `datainsert` and `fastinsert` functions will be removed in a future release. When using the `sqlwrite` function, you no longer have to preprocess or convert the data, as required by the `datainsert` function. The following short examples show how to append the same data using both the `sqlwrite` and `datainsert` functions. Use these examples for migrating to the `sqlwrite` function for data insertion.

Append data to an existing database table by using the `sqlwrite` function.

```
% Read from 'airlinesmall.csv'
impObj = detectImportOptions('airlinesmall.csv');
impObj = setvartype(impObj, ...
    {'DepTime', 'ArrTime', 'ActualElapsedTime', 'CRSElapsedTime', ...
    'ArrDelay', 'DepDelay', 'Distance'}, 'double');

airlines_data = readtable('airlinesmall.csv', impObj);

% Insert using sqlwrite function
sqlwrite(conn, 'airlinesmall', airlines_data);
```

Append the same data to the database table by using the `datainsert` function.

```
% Read from 'airlinesmall.csv'
impObj = detectImportOptions('airlinesmall.csv');
impObj = setvartype(impObj, ...
    {'DepTime', 'ArrTime', 'ActualElapsedTime', 'CRSElapsedTime', ...
    'ArrDelay', 'DepDelay', 'Distance'}, 'double');

airlines_data = readtable('airlinesmall.csv', impObj);
variablenames = airlines_data.Properties.VariableNames;
airlines_data = table2cell(airlines_data);

% Convert to compatible data
columns = size(airlines_data, 2);
for i = 1:columns
    a = airlines_data(:, i);
    if all(cellfun(@(x) isnumeric(x), a)) == true
        a(cellfun(@isnan, a)) = {Inf};
        airlines_data(:, i) = a;
    end
end

airlines_data = cell2table(airlines_data, 'VariableNames', variablenames);

% Insert using datainsert function
datainsert(conn, 'airlinesmall', variablenames, airlines_data);
```

When using the `datainsert` function, you must complete additional steps to preprocess the data to insert. Use the `sqlwrite` function instead to avoid these extra steps.

See Also

`sqlwrite` | `detectImportOptions` | `setvartype` | `readtable` | `table2cell` | `cell2table`

More About

- “Insert Data into New Database Table Using Insert Functionality” on page 5-55
- “Insert Data into Database Table” on page 5-61
- “Writing Data Common Errors” on page 3-2

Insert Data into New Database Table Using Insert Functionality

To insert data into a new database table, you can use the `sqlwrite` function. The `datainsert` and `fastinsert` functions will be removed in a future release. When using the `sqlwrite` function, you no longer have to preprocess or convert the data, as required by the `datainsert` function. The following short examples show how to insert the same data using both the `sqlwrite` and `datainsert` functions. Use these examples for migrating to the `sqlwrite` function for data insertion.

Insert data in a new database table by using the `sqlwrite` function.

```
% Read from patient.xls file
patient_data = readtable('patient.xls');

% Insert using sqlwrite function
sqlwrite(conn, 'patient', patient_data);
```

Insert the same data by using the `datainsert` function.

```
% Create a database table equivalent to data stored in patients.xls file
sqlquery = ['CREATE TABLE patients(LastName varchar, Gender varchar, ' ...
           'Age numeric, Location varchar, Height numeric, Weight numeric, ' ...
           'Smoker Boolean, Systolic numeric, Diastolic numeric, ' ...
           'SelfAssessedHealthStatus varchar)'];
execute(conn, sqlquery)

% Read from patients.csv file
patient_data = readtable('patients.csv');
variablenames = patient_data.Properties.VariableNames;
patient_data = table2cell(patient_data);

% Convert to compatible data
columns = size(patient_data,2);
for i = 1:columns
    a = patient_data(:,i);
    if all(cellfun(@(x)isnumeric(x),a)) == true
        a(cellfun(@isnan,a)) = {Inf};
        patient_data(:,i) = a;
    end
end

patient_data = cell2table(patient_data, 'VariableNames', variablenames);

% Insert using datainsert function
datainsert(conn, 'patient', variablenames, patient_data)
```

When using the `datainsert` function, you must complete additional steps to preprocess the data to insert. Use the `sqlwrite` function instead to avoid these extra steps.

See Also

`sqlwrite` | `readtable` | `table2cell` | `cell2table` | `exec`

More About

- “Append Data to Existing Database Table Using Insert Functionality” on page 5-53

- “Insert Data into Database Table” on page 5-61
- “Writing Data Common Errors” on page 3-2

Join Tables Using Command Line

You can join data in database tables and import the results interactively by using the **Database Explorer** app. Or, you can join two database tables by using the `sqlinnerjoin` and `sqlouterjoin` command line functions. The following short examples show how to join tables using the command line.

Enter this code to create an inner join between two database tables (the left table and right table of the join). An inner join retrieves records that have matching values in the shared column of both tables.

```
lefttable = 'productTable';
righttable = 'suppliers';
data = sqlinnerjoin(conn,lefttable,righttable);
```

To create an outer join, enter this code at the command line. An outer join retrieves the matched and unmatched rows between the two tables. This code shows a right outer join that uses different left and right keys in the corresponding database tables.

```
lefttable = 'employees';
righttable = 'departments';
data = sqlouterjoin(conn,lefttable,righttable, ...
    'LeftKey','MANAGER_ID','RightKey','DEPT_MANAGER_ID', ...
    'Type','right');
```

To join more than two database tables at a time, use the Database Explorer app. For details, see “Join Tables Using Database Explorer App” on page 4-10. Or, you can create and run an SQL script using the `executeSQLScript` function.

See Also

`sqlouterjoin` | `sqlinnerjoin`

More About

- “Join Tables Using Database Explorer App” on page 4-10
- “Importing Data Common Errors” on page 3-3

Import Data from Database Table Using `sqlread` Function

This example shows how to import data from a table in a Microsoft® Access™ database into the MATLAB® workspace using the `sqlread` function. The example then shows how to use an SQL script to import data from an SQL query that contains multiple joins.

Connect to Database

Create a Microsoft Access database connection with the data source name `dbdemo` using an ODBC driver and a blank user name and password. This database contains the table `producttable`.

```
conn = database('dbdemo', '', '');
```

If you are connecting to a database using a JDBC connection, then specify a different syntax for the database function.

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Import Data from Database Table

Import product data from the database table `producttable` by using the `sqlread` function and the database connection. This function imports data as a MATLAB table.

```
tablename = 'producttable';
data = sqlread(conn, tablename);
```

Display the product number and description in the imported data.

```
data(:, [1 5])
```

```
ans =
```

```
10×2 table
```

productnumber	productdescription
9	'Victorian Doll'
8	'Train Set'
7	'Engine Kit'
2	'Painting Set'
4	'Space Cruiser'
1	'Building Blocks'
5	'Tin Soldier'
6	'Sail Boat'
3	'Slinky'
10	'Teddy Bear'

Import Data Using Multiple Joins in SQL Query

Create an SQL script file named `salesvolume.sql` with the following SQL query. This SQL query uses multiple joins to join these tables in the `dbdemo` database:

- `producttable`
- `salesvolume`
- `suppliers`

The purpose of the query is to import sales volume data for suppliers located in the United States.

```
SELECT salesvolume.january
, salesvolume.february
, salesvolume.march
, salesvolume.april
, salesvolume.may
, salesvolume.june
, salesvolume.july
, salesvolume.august
, salesvolume.september
, salesvolume.october
, salesvolume.november
, salesvolume.december
, suppliers.country
FROM ((producttable
INNER JOIN salesvolume
ON producttable.stocknumber = salesvolume.stocknumber)
INNER JOIN suppliers
ON producttable.suppliernumber = suppliers.suppliernumber)
WHERE suppliers.country LIKE 'United States%'
```

Run the `salesvolume.sql` file by using the `executeSQLScript` function. `results` is a structure array with the data returned from running the SQL query in the SQL script file.

```
results = executeSQLScript(conn, 'salesvolume.sql');
```

Display the first three rows in the `Data` table. Access this table as a field of the structure array by using dot notation.

```
head(results(1).Data,3)
```

```
ans =
```

```
3×13 table
```

january	february	march	april	may	june	july	august	september	october
5000	3500	2800	2300	1700	1400	1000	900	1600	3300
2400	1721	1414	1191	983	825	731	653	723	790
1200	900	800	500	399	345	300	175	760	1500

Close Database Connection

`close(conn)`

See Also

`executeSQLScript` | `sqlread` | `database` | `close`

More About

- “Data Import Memory Management” on page 5-20

Insert Data into Database Table

This example shows how to import data from a database into MATLAB®, perform calculations on the data, and export the results to a database table.

The example assumes that you are connecting to a Microsoft® Access™ database that contains tables named `salesvolume` and `yearlysales`. Also, the example assumes that you start MATLAB as an administrator. The `salesvolume` table contains the column names for each month. The `yearlysales` table contains the column names `month` and `salestotal`.

Connect to Database

Create a database connection to the Microsoft Access database. For example, this code assumes that you are connecting to a data source named `dbdemo` with a blank user name and password.

```
conn = database('dbdemo', '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Calculate Sum of Sales Volume for One Month

Import sales volume data for the month of March using the database connection. The `salesvolume` database table contains sales volume data.

```
tablename = 'salesvolume';
data = sqlread(conn, tablename);
```

Display the first three rows of sales volume data. The fourth variable contains the data for the month of March.

```
head(data(:,4),3)
```

```
ans =
```

```
 3×1 table
```

```
  march
```

```
  _____
```

```
  981
```

```
 1414
```

```
 890
```

Calculate the sum of the March sales. Assign the result to the MATLAB workspace variable `total` and display the result.

```
total = sum(data.march)
```

```
total =  
    14606
```

Insert Total Sales for One Month into Database

Retrieve the name of the month from the sales volume data.

```
month = data.Properties.VariableNames(4);
```

Define the names of the columns for the data to insert as a cell array of character vectors.

```
colnames = {'month' 'salestotal'};
```

Create a MATLAB table that stores the data to export.

```
results = table(month,total,'VariableNames',colnames);
```

Determine the status of the AutoCommit database flag. This status determines whether or not the insert action can be undone.

```
conn.AutoCommit
```

```
ans =  
    'on'
```

The AutoCommit flag is set to on. The database commits the exported data automatically to the database, and this action cannot be undone.

Insert the sum of sales for the month of March into the yearllysales table.

```
tablename = 'yearllysales';  
sqlwrite(conn,tablename,results)
```

Import the data from the yearllysales table. This data contains the calculated result.

```
data = sqlread(conn,tablename)
```

```
data =  
    1×3 table  
    month    salestotal    revenue  
    _____    _____    _____  
    'march'    14606        0
```


Close Database Connection

`close(conn)`

See Also

`sqlwrite` | `sqlread` | `database` | `close`

More About

- “Export Data Using Bulk Insert” on page 5-13

Retrieve Database Metadata

This example shows how to retrieve database information using the `connection` object and the `sqlfind` function.

The example assumes that you are connecting to a MySQL® database that contains a table named `productTable`.

Connect to Database

Create an ODBC database connection to a MySQL database with a user name and password.

```
datasource = "MySQL ODBC";  
username = "username";  
password = "password";  
conn = database(datasource,username,password);
```

Find Catalogs and Schemas

Display the catalogs in the database by using the `Catalogs` property of the `connection` object.

```
conn.Catalogs
```

```
ans =  
  
1×7 cell array  
  
Columns 1 through 4  
    {'information_sch...'}    {'detsdb'}    {'mysql'}    {'performance_sch...'}  
  
Columns 5 through 7  
    {'sys'}    {'toy_store'}    {'toystore_doc'}
```

Display the schemas in the database by using the `Schemas` property of the `connection` object.

```
conn.Schemas
```

```
ans =  
  
0×0 empty cell array
```

Find Table Types

Find all table types in the database by using the `sqlfind` function with the `connection` object.

```
tables = sqlfind(conn, '');
```

Display the first three table types.

```
tables(1:3,:)
```

```
ans =
```

```
3x5 table
```

Catalog	Schema	Table	Columns	Type
{'toystore_doc'}	{0x0 char}	{'Person' }	{1x5 cell}	{'TABLE'}
{'toystore_doc'}	{0x0 char}	{'airlinesmall' }	{1x29 cell}	{'TABLE'}
{'toystore_doc'}	{0x0 char}	{'inventoryTable'}	{1x4 cell}	{'TABLE'}

Find the table type of the table productTable.

```
tablename = 'productTable';
data = sqlfind(conn,tablename);
data.Type
```

```
ans =
```

```
1x1 cell array
```

```
{'TABLE'}
```

Find Table Columns

Find all columns in the database table productTable and display them.

```
data = sqlfind(conn,tablename);
data.Columns{:}
```

```
ans =
```

```
1x5 cell array
```

```
Columns 1 through 4
```

```
{'productNumber'} {'stockNumber'} {'supplierNumber'} {'unitCost'}
```

```
Column 5
```

```
{'productDescript...'}

```

Close Database Connection

```
close(conn)
```

See Also

sqlread | sqlfind | database | close

More About

- “Import Data from Database Table Using sqlread Function” on page 5-58

Customize Options for Importing Data from Database into MATLAB

This example shows how to customize import options when importing data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options for multiple database columns. Import data using the `sqlread` function.

The example uses the `patients.xls` spreadsheet, which contains patient information. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create Database Connection

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load Example Data

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';
sqlwrite(conn, tablename, patients)
```

Create SQLImportOptions Object

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn, tablename);
```

Display the default data types of the variables by accessing the `VariableNames` and `VariableTypes` properties of the `SQLImportOptions` object using dot notation.

```
disp([opts.VariableNames' opts.VariableTypes'])
```

```
'LastName'          'char'
'Gender'            'char'
'Age'              'double'
'Location'         'char'
'Height'           'double'
'Weight'           'double'
'Smoker'           'double'
'Systolic'         'double'
'Diastolic'        'double'
'SelfAssessedHealthStatus' 'char'
```

Customize Import Options

Change the data types of multiple variables. Convert the data type for all text variables to `string`. Also, convert the data type for all numeric variables to `single`.

```
textvars = {'LastName', 'Gender', 'Location', 'SelfAssessedHealthStatus'};
opts = setoptions(opts, textvars, 'Type', 'string');

numvars = {'Age', 'Height', 'Weight', 'Systolic', 'Smoker', 'Diastolic'};
opts = setoptions(opts, numvars, 'Type', 'single');
```

Display the updated data types of the variables.

```
disp([opts.VariableNames' opts.VariableTypes'])

    'LastName'          'string'
    'Gender'           'string'
    'Age'              'single'
    'Location'         'string'
    'Height'           'single'
    'Weight'           'single'
    'Smoker'           'single'
    'Systolic'         'single'
    'Diastolic'        'single'
    'SelfAssessedHealthStatus' 'string'
```

Set the import options to replace missing data in the specified variables with the fill value `unknown`.

```
varnames = {'LastName', 'Location'};
opts = setoptions(opts, varnames, 'FillValue', "unknown");
```

Set the import options to change the text of the `SelfAssessedHealthStatus` variable to lowercase.

```
varname = {'SelfAssessedHealthStatus'};
opts = setoptions(opts, varname, 'TextCaseRule', "lower");
```

Set the import options to omit rows with missing data in the `LastName` variable.

```
varname = {'LastName'};
opts = setoptions(opts, varname, 'MissingRule', "omitrow");
```

Preview Data Before Importing

Before importing the data, preview it by using the customized import options.

```
T = preview(opts)
```

```
T=8x10 table
    LastName      Gender      Age      Location      Height      Weight      Smoker
    _____  _____  _____  _____  _____  _____  _____
    "Smith"       "Male"     38      "County General Hospital"  71          176          1
    "Johnson"    "Male"     43      "VA Hospital"  69          163          0
    "Williams"   "Female"   38      "St. Mary's Medical Center"  64          131          0
    "Jones"      "Female"   40      "VA Hospital"  67          133          0
    "Brown"      "Female"   49      "County General Hospital"  64          119          0
    "Davis"      "Female"   46      "St. Mary's Medical Center"  68          142          0
    "Miller"     "Female"   33      "VA Hospital"  64          142          1
```

```
"Wilson"      "Male"      40      "VA Hospital"      68      180      0
```

Import Data Using Import Options

Import the variables with the customized data types by using the `sqlread` function, and display the first eight rows of imported data.

```
T = sqlread(conn,tablename,opts);
head(T)
```

ans=8×10 table

LastName	Gender	Age	Location	Height	Weight	Smoker
"Smith"	"Male"	38	"County General Hospital"	71	176	1
"Johnson"	"Male"	43	"VA Hospital"	69	163	0
"Williams"	"Female"	38	"St. Mary's Medical Center"	64	131	0
"Jones"	"Female"	40	"VA Hospital"	67	133	0
"Brown"	"Female"	49	"County General Hospital"	64	119	0
"Davis"	"Female"	46	"St. Mary's Medical Center"	68	142	0
"Miller"	"Female"	33	"VA Hospital"	64	142	1
"Wilson"	"Male"	40	"VA Hospital"	68	180	0

Delete Example Data and Close Database Connection

Delete the `patients` database table using the `execute` function.

```
sqlquery = ['DROP TABLE ' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

See Also

[databaseImportOptions](#) | [setoptions](#) | [getoptions](#) | [preview](#) | [sqlread](#) | [database](#) | [close](#) | [sqlwrite](#) | [execute](#)

More About

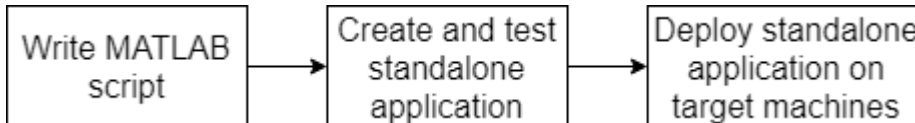
- "Importing Data Common Errors" on page 3-3

External Websites

- [SQL Tutorial](#)

Deploy Relational Database Application with MATLAB Compiler

This example shows how to write a script to analyze data stored in a relational database, and deploy the script as a standalone application. Write code that connects to a database, imports data from the database into MATLAB®, analyzes the data, and closes the database connection. Then, you can deploy the code by compiling it as a standalone application by using the Application Compiler (MATLAB Compiler) app and running the application on other machines.



The example uses a JDBC driver to create the database connection. For a JDBC driver, include the JDBC driver JAR file among the files installed with your application. For an ODBC driver, install the ODBC driver and configure an ODBC data source on each machine where you run the application. For details about configuring ODBC and JDBC drivers, see “Configure Driver and Data Source” on page 2-14.

Overall, the example follows the steps described in “Create Standalone Application from MATLAB Function” (MATLAB Compiler) and updates the steps for a standalone database application.

Ensure that you have administrator privileges on the other machines to run the standalone application.

Create Function in MATLAB

Write a MATLAB script named `importAndAnalyzeDataFromDatabase.m` and save it in a file location of your choice. The script contains the `importAndAnalyzeDataFromDatabase` function, which returns the maximum product number from the data in the `productTable` database table. The function connects to a Microsoft® SQL Server® database and imports all data from `productTable`. Then, the function calculates the maximum product number.

type `importAndAnalyzeDataFromDatabase.m`

```

function maxProdNum = importAndAnalyzeDataFromDatabase
% IMPORTANDANALYZEDATAFROMDATABASE The importAndAnalyzeDataFromDatabase
% function connects to a Microsoft® SQL Server® database using a JDBC
% driver, imports data from the database into MATLAB®, performs a simple
% data analysis, and closes the database connection.

%%
% Connect to the database by using the |Vendor| name-value pair argument of
% the database function to specify a connection to an |SQLServer| database.
% Set the |AuthType| name-value pair argument to |Server|. For example,
% this code assumes that you are connecting to a database named |dbname|,
% on a database server named |sname|, with the user name |username|, the
% password |pwd|, and the port number |123456|.
conn = database('dbname','username','pwd', ...
    'Vendor','Microsoft SQL Server','Server','sname', ...
    'AuthType','Server','PortNumber',123456);
%%
% Import data from the |productTable| database table.
tablename = 'productTable';
  
```



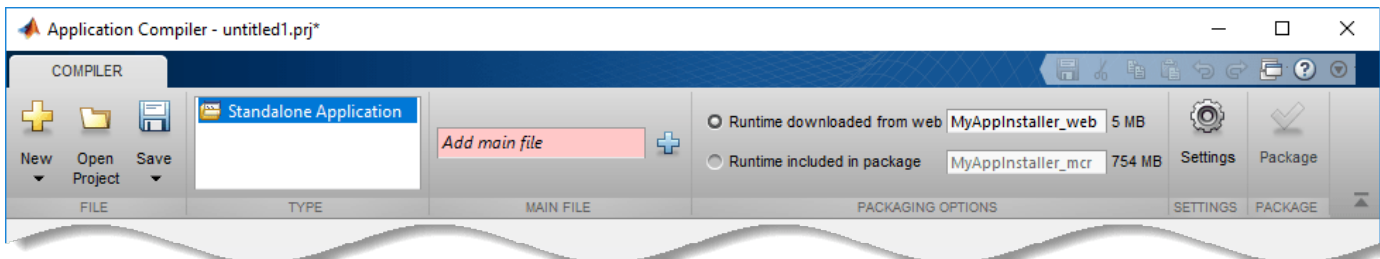
```

data = sqlread(conn,tablename);
%%
% Determine the highest product number among products.
prodNums = data.productnumber;
maxProdNum = max(prodNums);
%%
% Close the database connection.
close(conn)
end


```

Create Standalone Application Using Application Compiler App

On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow to open the apps gallery. Under **Application Deployment**, click **Application Compiler**.



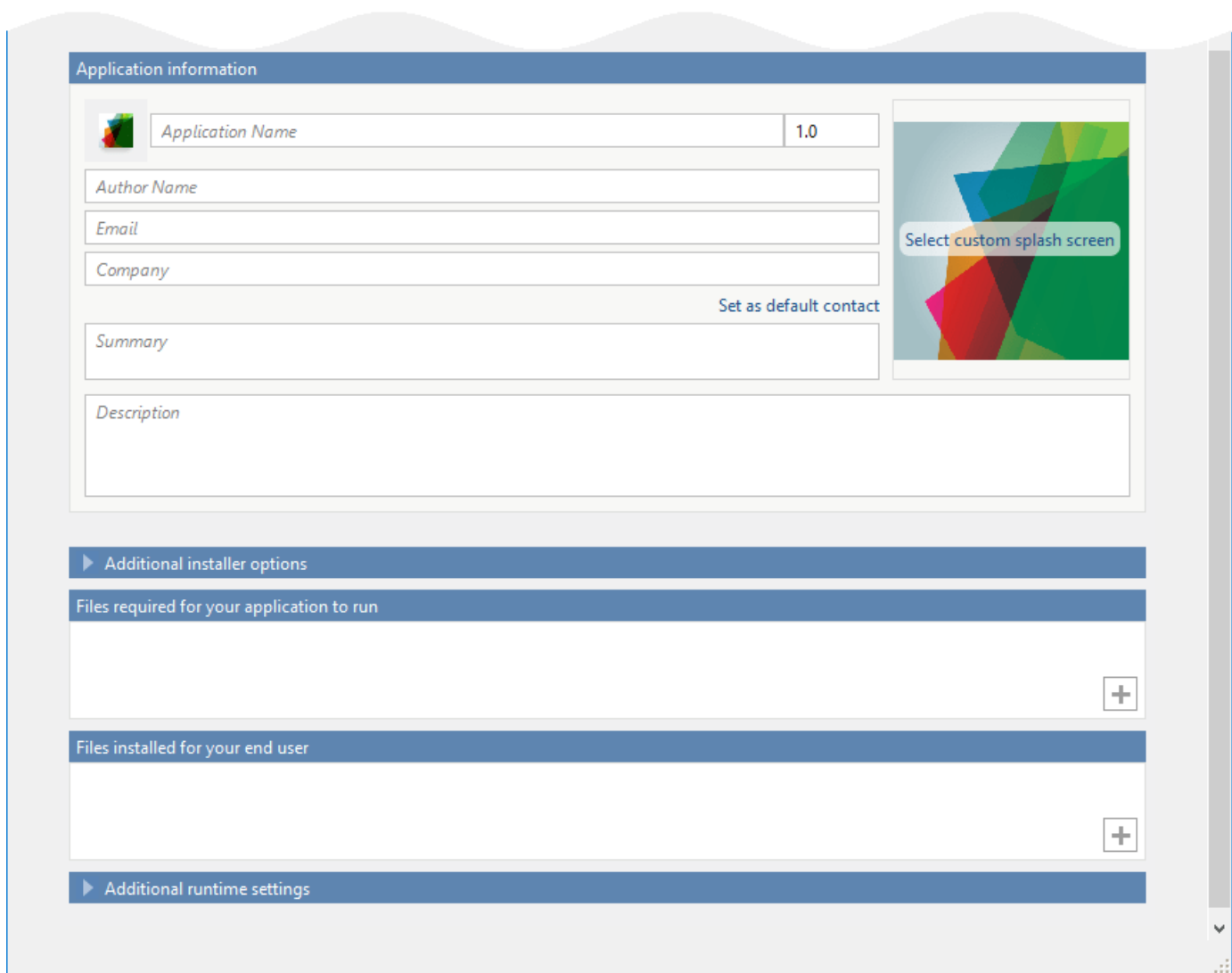
In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.

- 1 In the **Main File** section of the toolbar, click .
- 2 In the **Add Files** dialog box, browse to the file location that contains your saved script. Select `importAndAnalyzeDataFromDatabase.m` and click **Open**. The Application Compiler app adds the `importAndAnalyzeDataFromDatabase` function to the list of main files.

Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:

- **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application
- **Runtime included in package** — Generates an installer that includes the MATLAB Runtime installer

Customize the packaged application and its appearance by entering the following options:



- **Application information** — Editable information about the deployed application. You can also customize the appearance of the standalone application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata.
- **Additional installer options** — Options for editing the default installation path for the generated installer and selecting a custom logo.
- **Files required for your application to run** — Additional files required by the generated application to run. The software includes these files in the generated application installer.
- **Files installed for your end user** — Files that are installed with your application. These files include the generated `readme.txt` file and the generated executable for the target platform. *To create a database connection using the standalone application, add the driver JAR file.* In this case, add `sqljdbc4.jar`.
- **Additional runtime settings** — Platform-specific options for controlling the generated executable.

For details about these options, see “Customize an Application” (MATLAB Compiler).

To generate the packaged application, click **Package** in the **Package** section on the toolbar. In the Save Project dialog box, specify the location in which to save the project.

In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output.

- `for_redistribution` — Folder containing the file that installs the application and the MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc` (such as binary, JAR, header, and source files for a specific target). Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler™.

Install and Run Standalone Application

To install the standalone application, in the `for_redistribution` folder, double-click the `MyAppInstaller_web` executable.

If you want to connect to the Internet using a proxy server, click **Connection Settings**. Enter the proxy server settings in the provided dialog box. Click **OK**.

To complete the installation, follow the instructions in the installation wizard.

To run your standalone application:

- 1 Open a terminal window.
- 2 Navigate to the folder in which you installed the application.
- 3 Run the application.

Test Standalone Application on Target Machine

Choose one target machine to test the MATLAB generated standalone application.

Copy the files in the `for_testing` folder to the target machine.

To test your standalone application:

- 1 Open a terminal window.
- 2 Navigate to the `for_testing` folder.
- 3 Run the application.

Deploy Standalone Application on Target Machines

Copy the `for_redistribution_files_only` folder to a file location on all target machines where MATLAB or MATLAB Runtime is installed.

Run the MATLAB generated standalone application on all target machines by using the executable in the `for_redistribution_files_only` folder.

See Also

`database` | `close` | `sqlread`

More About

- “Create Functions in Files”
- “Create Standalone Application from MATLAB Function” (MATLAB Compiler)
- “Customize an Application” (MATLAB Compiler)
- “Import Data from Database Table Using `sqlread` Function” on page 5-58

Import Data Using SQL Prepared Statement with Multiple Parameter Values

This example shows how to import data from a Microsoft® SQL Server® database using an SQL prepared statement with a JDBC database connection. Use the SELECT SQL statement in a loop to execute the same SQL query for multiple values. Import the data from the database and display the results.

The SQL prepared statement is a database feature that enables you to execute the same SQL statement repeatedly with high efficiency. As you define the SQL prepared statement and bind values to parameters, the database completes these actions:

- Create an SQL statement template with parameters.
- Parse, compile, and perform query optimization on the SQL statement template, and store the results without execution.
- Bind values to the parameters and execute the SQL statement. (An application can execute the statement as many times as specified with different values.)

The advantages of using SQL prepared statements include improved performance and security.

You can execute SQL prepared statements by using a JDBC database connection only.

Connect to Database

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
conn = database(datasource, '', '');
```

Create SQL Prepared Statement

Create an SQL prepared statement for importing data from the SQL Server database using the JDBC database connection. The question marks in the SELECT SQL statement indicate it is an SQL prepared statement. This statement selects all data from the database table `productTable` for specified product descriptions.

```
query = strcat("SELECT * FROM productTable ", ...
    "WHERE productDescription = ?");
pstmt = databasePreparedStatement(conn, query);
```

Bind Multiple Values and Execute SQL Prepared Statement

Select the single parameter in the SQL prepared statement using its numeric index. Specify the values to bind as a string array containing three product descriptions: train set, engine kit, and slinky.

```
selection = [1];
values = ["Train Set" "Engine Kit" "Slinky"];
```

Bind parameter values in the SQL prepared statement. Using a for loop, bind the values for each product description and import data from the database using the bound parameter values. The results contain a table with three rows of data for the products with the specified product descriptions.

```
for i = 1:3
    pstmt = bindParamValues(pstmt,selection,values(i));
    results(i,:) = fetch(conn,pstmt);
end
results
```

```
results=3x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
  _____  _____  _____  _____  _____
           8      2.1257e+05      1001           5      {'Train Set' }
           7      3.8912e+05      1007           16      {'Engine Kit'}
           3      4.01e+05      1009           17      {'Slinky'   }
```

Close SQL Prepared Statement and Database Connection

```
close(pstmt)
close(conn)
```

See Also

[database](#) | [close](#) | [databasePreparedStatement](#) | [bindParamValues](#) | [close](#) | [fetch](#)

More About

- “SQL Prepared Statement Error Messages” on page 3-15

MySQL Native Interface Topics

Configure MySQL Native Interface Data Source

This tutorial shows how to set up a MySQL native interface data source and connect to a MySQL database using the Database Explorer app or the command line. This tutorial uses the MariaDB® C Connector driver to connect to a MySQL database.

The configuration steps show configuring the MySQL native data source on the Windows platform. The same steps work on Linux and macOS platforms.

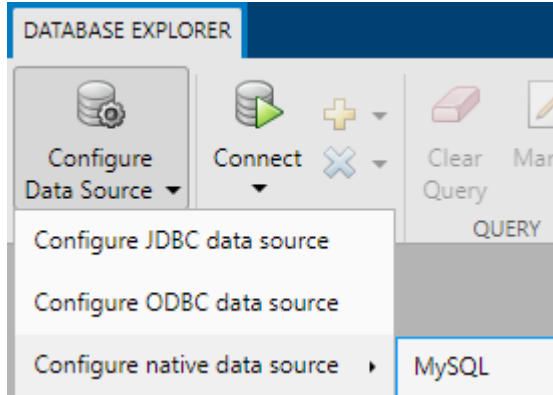
Database Toolbox provides MariaDB C Connector driver for connecting to MySQL server. Therefore, you do not have to install and configure the driver.

Step 1. Set up the data source.

You can set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure native data source > MySQL**.



The MySQL Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter the name of your data source. (This example uses a data source named `MySQLDataSource`.) You use this name to establish a connection to your database.

The screenshot shows the 'MySQL Data Source Configuration' dialog box. It is divided into three sections:

- Data Source Details:** A text box for 'Name' containing 'MySQLDataSource'.
- Connection Parameters:** Three text boxes: 'Database Name' (empty), 'Server' (localhost), and 'Port Number' (3306).
- Connection Options:** A table with two columns: 'Name' and 'Value'. The table has three rows numbered 1, 2, and 3. To the left of the table are '+' and '-' buttons. Below the table are 'Edit', 'Test', 'Save', and 'Delete' buttons.

At the bottom of the dialog is a 'Message' text area.

- 4 In the **Database Name** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.
- 5 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign + to specify additional driver-specific options.
- 6 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 7 Click **Save**. The MySQL Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a MySQL native interface data source for a MySQL database.

```
vendor = "MySQL";
opts = databaseConnectionOptions("native", vendor);
```

- 2 Set the database connection options. For example, this code assumes that you are connecting to a data source named MySQLDataSource, database name toystore_doc, database server dbtb01, and port number 3306.

```
opts = setoptions(opts, ...
  'DataSourceName', "MySQLDataSource", ...
  'DatabaseName', "toystore_doc", 'Server', "dbtb01", ...
  'PortNumber', 3306);
```

- 3 Test the database connection by specifying the user name and password.

```
username = "root";  
password = "matlab";  
status = testConnection(opts,username,password);
```

- 4 Save the data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the MySQL database using the Database Explorer app or the command line.

Step 4. Connect using the Database Explorer app or the command line.

Connect to MySQL Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 In the **Catalog** list, select the catalog. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to MySQL Using Command Line

- 1 Connect to a MySQL database using the configured MySQL native interface data source, user name, and password.

```
datasource = "MySQLDataSource";  
username = "root";  
password = "matlab";  
conn = mysql(datasource,username,password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps
Database Explorer

Functions

mysql | close | databaseConnectionOptions | saveAsDataSource | setoptions |
testConnection | setenv

More About

- “Import Data from MySQL Database Table” on page 6-6
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13

Import Data from MySQL Database Table

This example shows how to import data from a table in a MySQL® database into the MATLAB® workspace using the `sqlread` and `fetch` functions with the MySQL native interface.

Connect to Database

Create a MySQL native interface database connection using the data source name `MySQLDataSource` and a user name and password. The MySQL database contains the table `productTable`.

```
datasource = "MySQLDataSource";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Import Data from Database Table Using `sqlread` Function

Import product data from the database table `productTable` by using the `sqlread` function and the database connection. This function imports data as a MATLAB table.

```
tablename = "productTable";
data = sqlread(conn,tablename);
```

Display the product number and description in the imported data.

```
data(:,[1 5])
```

```
ans=10x2 table
   productNumber  productDescription
   _____  _____
           9      "Victorian Doll"
           8      "Train Set"
           7      "Engine Kit"
           2      "Painting Set"
           4      "Space Cruiser"
           1      "Building Blocks"
           5      "Tin Soldier"
           6      "Sail Boat"
           3      "Slinky"
          10      "Teddy Bear"
```

Import Data from Database Table Using `fetch` Function

Import product data from the database table `productTable` by using the `fetch` function and the database connection. Create an SQL query to import data that is sorted by product description alphabetically. The `fetch` function imports data as a MATLAB table.

```
sqlquery = "SELECT * FROM productTable ORDER BY productDescription ASC";
data = fetch(conn,sqlquery);
```

Display the product number and description in the imported data.

```
data(:,[1 5])
```

```
ans=10x2 table
  productNumber  productDescription
  _____  _____
         1      "Building Blocks"
         7      "Engine Kit"
         2      "Painting Set"
         6      "Sail Boat"
         3      "Slinky"
         4      "Space Cruiser"
        10      "Teddy Bear"
         5      "Tin Soldier"
         8      "Train Set"
         9      "Victorian Doll"
```

Close Database Connection

```
close(conn)
```

See Also

`mysql` | `close` | `sqlread` | `executeSQLScript`

More About

- “Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface” on page 6-8

External Websites

- [MySQL Documentation](#)

Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface

This example shows how to customize import options when importing data from a database table using the MySQL® native interface. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options for multiple database columns. Import data using the `sqlread` function.

The example uses the `patients.xls` spreadsheet, which contains patient information. Also, the example uses a MySQL database with the MariaDB® C Connector driver.

Create Database Connection

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password.

```
datasource = "MySQLDataSource";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Load Example Data

Load patient information into the MATLAB® workspace.

```
patients = readtable("patients.xls");
```

Create the `patients` database table using the patient information.

```
tablename = "patients";
sqlwrite(conn,tablename,patients)
```

Create SQLImportOptions Object

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename);
```

Display the default data types of the variables by accessing the `VariableNames` and `VariableTypes` properties of the `SQLImportOptions` object using dot notation.

```
disp([opts.VariableNames' opts.VariableTypes'])

    {'LastName'          }    {'string' }
    {'Gender'           }    {'string' }
    {'Age'              }    {'double' }
    {'Location'         }    {'string' }
    {'Height'           }    {'double' }
    {'Weight'           }    {'double' }
    {'Smoker'           }    {'logical' }
    {'Systolic'         }    {'double' }
    {'Diastolic'        }    {'double' }
    {'SelfAssessedHealthStatus'} {'string' }
```

Customize Import Options

Change the data types of multiple variables. Convert the data type for all text variables to `string`. Also, convert the data type for all numeric variables to `single`.

```
textvars = ["LastName" "Gender" "Location" "SelfAssessedHealthStatus"];
opts = setoptions(opts,textvars,'Type','string');

numvars = ["Age" "Height" "Weight" "Systolic" "Smoker" "Diastolic"];
opts = setoptions(opts,numvars,'Type','single');
```

Display the updated data types of the variables.

```
disp([opts.VariableNames' opts.VariableTypes'])

{'LastName'          } {'string'}
{'Gender'            } {'string'}
{'Age'               } {'single'}
{'Location'          } {'string'}
{'Height'            } {'single'}
{'Weight'            } {'single'}
{'Smoker'            } {'single'}
{'Systolic'          } {'single'}
{'Diastolic'         } {'single'}
{'SelfAssessedHealthStatus' } {'string'}
```

Set the import options to replace missing data in the specified variables with the fill value `unknown`.

```
varnames = ["LastName" "Location"];
opts = setoptions(opts,varnames,'FillValue','unknown');
```

Set the import options to omit rows with missing data in the `LastName` variable.

```
varname = "LastName";
opts = setoptions(opts,varname,'MissingRule','omitrow');
```

Preview Data Before Importing

Before importing the data, preview it by using the customized import options.

```
T = preview(opts)
```

```
T=8x10 table
  LastName      Gender      Age      Location      Height      Weight      Smoker
  _____  _____  _____  _____  _____  _____  _____
  "Smith"      "Male"      38      "County General Hospital"      71      176      1
  "Johnson"    "Male"      43      "VA Hospital"      69      163      0
  "Williams"    "Female"    38      "St. Mary's Medical Center"    64      131      0
  "Jones"       "Female"    40      "VA Hospital"      67      133      0
  "Brown"       "Female"    49      "County General Hospital"    64      119      0
  "Davis"       "Female"    46      "St. Mary's Medical Center"    68      142      0
  "Miller"      "Female"    33      "VA Hospital"      64      142      1
  "Wilson"     "Male"      40      "VA Hospital"      68      180      0
```

Import Data Using Import Options

Import the variables with the customized data types by using the `sqlread` function, and display the first eight rows of imported data.

```
T = sqlread(conn,tablename,opts);  
head(T)
```

LastName	Gender	Age	Location	Height	Weight	Smoker
"Smith"	"Male"	38	"County General Hospital"	71	176	1
"Johnson"	"Male"	43	"VA Hospital"	69	163	0
"Williams"	"Female"	38	"St. Mary's Medical Center"	64	131	0
"Jones"	"Female"	40	"VA Hospital"	67	133	0
"Brown"	"Female"	49	"County General Hospital"	64	119	0
"Davis"	"Female"	46	"St. Mary's Medical Center"	68	142	0
"Miller"	"Female"	33	"VA Hospital"	64	142	1
"Wilson"	"Male"	40	"VA Hospital"	68	180	0

Delete Example Data and Close Database Connection

Delete the `patients` database table using the `execute` function.

```
sqlquery = strcat("DROP TABLE ",tablename);  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

See Also

[databaseImportOptions](#) | [mysql](#) | [close](#) | [sqlwrite](#) | [setoptions](#) | [preview](#) | [sqlread](#) | [execute](#)

More About

- “Import Data from MySQL Database Table” on page 6-6

External Websites

- [MySQL Documentation](#)

Import Large Data Using DatabaseDatastore Object and MySQL Native Interface

This example shows how to use the `databaseDatastore` function to create a `DatabaseDatastore` object for accessing collections of data stored in a MySQL® database using the MySQL native interface. After creating a `DatabaseDatastore` object, you can preview data, read data in chunks, and read every record in the data set.

To analyze large data, you can run algorithms on large data sets using a tall array. Alternatively, you can write a MapReduce algorithm that defines the chunking and reduction of the data.

This example uses a preconfigured MySQL data source to create the database connection. For more information, see the `databaseConnectionOptions` function.

Create DatabaseDatastore Object

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password.

```
datasource = "MySQLDataSource";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Create a `DatabaseDatastore` object using the database connection and an SQL query. This query retrieves all data from the `airlinesmall` table.

```
sqlquery = "select * from airlinesmall";
dbds = databaseDatastore(conn,sqlquery);
```

Preview Data in DatabaseDatastore Object

Preview the first eight records in the data set returned by executing the SQL query.

```
preview(dbds)
```

```
ans=8x29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1990	9	11	2	1810	1812	1939	1930
1990	10	27	6	1353	1355	1634	1640
1990	10	23	2	1057	1055	1205	1155
1990	10	8	1	1515	1440	1609	1535
1990	10	19	5	1130	1120	1203	1154
1990	10	12	5	1755	1733	1858	1820
2001	11	22	4	1345	1355	1530	1549
2001	11	26	1	2105	2110	2209	2237

Read Data in DatabaseDatastore Object

Read the first 10 records.

```
dbds.ReadSize = 10;
read(dbds)
```

```
ans=10x29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1990	9	11	2	1810	1812	1939	1930
1990	10	27	6	1353	1355	1634	1640
1990	10	23	2	1057	1055	1205	1155
1990	10	8	1	1515	1440	1609	1535
1990	10	19	5	1130	1120	1203	1154
1990	10	12	5	1755	1733	1858	1820
2001	11	22	4	1345	1355	1530	1549
2001	11	26	1	2105	2110	2209	2237
2001	11	18	7	1407	1415	1442	1457
2001	12	23	7	1327	1310	1530	1530

Read the DatabaseDatastore object two more times by using the counter n. Read 10 records at a time.

```
n = 0;
while(hasdata(dbds) && n~=2)
    read(dbds)
    n = n+1;
end
```

```
ans=10x29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
2001	12	8	6	1402	1410	1642	1626
2001	12	22	6	1707	1715	1823	1821
2001	12	9	7	656	650	824	823
2002	6	19	3	632	640	748	756
2002	6	10	1	927	930	1031	1031
2002	6	15	6	1120	1115	1401	1413
2002	6	8	6	1557	1600	1703	1711
2002	6	26	3	2026	1840	47	2257
2002	6	5	3	2032	2031	2233	2248
2002	7	25	4	1032	1035	1853	1852

```
ans=10x29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	30	5	1329	1329	1434	1436
1987	11	7	6	1316	1315	1713	1647
1987	11	28	6	815	815	1015	1015
1987	11	2	1	700	700	800	800
1987	11	14	6	840	840	1127	1120
1987	11	1	7	1625	1625	1823	1758
1987	11	26	4	1314	1315	1538	1542
1992	7	18	6	1538	1540	1703	1720
1992	7	19	7	932	932	1130	1052
1992	8	4	2	NaN	1815	NaN	1940

Reset DatabaseDatastore Object

Reset the DatabaseDatastore object to its original state, where no data has been read from it. Resetting allows you to reread from the same DatabaseDatastore object.

```
reset(dbds)
```

Read Every Record in DatabaseDatastore Object

Read every record in the DatabaseDatastore object in increments of 50,000 records at a time.

```
dbds.ReadSize = 50000;
data = readall(dbds);
```

Display the first three records of the full data set.

```
head(data,3)
```

```
ans=3x29 table
   Year   Month   DayofMonth   DayOfWeek   DepTime   CRSDepTime   ArrTime   CRSArrTime
   _____
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1990	9	11	2	1810	1812	1939	1930
1990	10	27	6	1353	1355	1634	1640
1990	10	23	2	1057	1055	1205	1155

Close DatabaseDatastore Object and Database Connection

```
close(dbds)
```

See Also

[mysql](#) | [close](#) | [hasdata](#) | [preview](#) | [read](#) | [readall](#) | [databaseDatastore](#)

More About

- “Import Data from MySQL Database Table” on page 6-6

Insert Data into Database Table Using MySQL Native Interface

This example shows how to import data from a database into MATLAB®, perform calculations on the data, and export the results to a database table.

The example assumes that you are connecting to a MySQL® database that contains tables named `salesVolume` and `yearlySales`. Also, the example uses a MySQL database with the MariaDB® C Connector driver. The `salesVolume` table contains the column names for each month. The `yearlySales` table contains the column names `Month` and `SalesTotal`.

Create Database Connection

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password.

```
datasource = "MySQLDataSource";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Calculate Sum of Sales Volume for One Month

Import sales volume data for the month of March using the database connection. The `salesVolume` database table contains sales volume data.

```
tablename = "salesVolume";
data = sqlread(conn,tablename);
```

Display the first three rows of sales volume data. The fourth variable contains the data for the month of March.

```
head(data(:,4),3)
```

```
ans=3x1 table
    March
-----
    981
   1414
    890
```

Calculate the sum of the March sales. Assign the result to the MATLAB workspace variable `total` and display the result.

```
total = sum(data.March)
```

```
total = 14606
```

Insert Total Sales for One Month into Database

Retrieve the name of the month from the sales volume data.

```
month = data.Properties.VariableNames(4);
```

Define the names of the columns for the data to insert as a string array.

```
colnames = ["Month" "SalesTotal"];
```

Create a MATLAB table that stores the data to export.

```
results = table(month,total,'VariableNames',colnames);
```

Determine the status of the AutoCommit database flag. This status determines whether or not the insert action can be undone.

```
conn.AutoCommit
```

```
ans =  
"on"
```

The AutoCommit flag is set to on. The database commits the exported data automatically to the database, and this action cannot be undone.

Insert the sum of sales for the month of March into the yearlySales table using the toystore_doc catalog.

```
tablename = "yearlySales";  
sqlwrite(conn,tablename,results,'Catalog','toystore_doc')
```

Import the data from the yearlySales table. This data contains the calculated result.

```
data = sqlread(conn,tablename)
```

```
data=1x2 table  
    Month    SalesTotal  
-----  
    "March"    14606
```

Close Database Connection

```
close(conn)
```

See Also

mysql | close | sqlread | sqlwrite

More About

- “Import Data from MySQL Database Table” on page 6-6
- “Delete Data from Database Using MySQL Native Interface” on page 6-19
- “Roll Back Data in Database Using MySQL Native Interface” on page 6-16

Roll Back Data in Database Using MySQL Native Interface

This example shows how to connect to a database, insert a row into an existing database table, and roll back the insert using the MySQL® native interface. The example uses a MySQL database with the MariaDB® C Connector driver. The database contains the table `productTable`.

Create Database Connection

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password.

```
datasource = "MySQLDataSource";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Append Data to Existing Database Table

Set the `AutoCommit` property of the connection object to `off`. Any updates you make after turning off this flag do not commit to the database automatically.

```
conn.AutoCommit = "off";
```

To view the existing database table `productTable` before appending data, import its contents into MATLAB® and display the last few rows.

```
tablename = "productTable";
rows = sqlread(conn,tablename);
tail(rows,3)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	4.7082e+05	1012	17	"Pancakes"
14	5.101e+05	1011	19	"Shawl"
15	8.9975e+05	1011	20	"Snacks"

Create a MATLAB table that contains the data for one product.

```
data = table(30,500000,1000,25,"Rubik's Cube", ...
    'VariableNames',["productnumber" "stocknumber" ...
    "suppliernumber" "unitcost" "productdescription"]);
```

Append the product data into the database table `productTable`.

```
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB again and display the last few rows. The results contain a new row for the inserted product.

```
rows = sqlread(conn,tablename);
tail(rows,4)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	4.7082e+05	1012	17	"Pancakes"

14	5.101e+05	1011	19	"Shawl"
15	8.9975e+05	1011	20	"Snacks"
30	5e+05	1000	25	"Rubik's Cube"

Roll Back Data

Roll back the inserted row.

```
rollback(conn)
```

Import the contents of the database table into MATLAB again and display the last few rows. The results no longer contain the inserted row.

```
rows = sqlread(conn,tablename);
tail(rows,3)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	4.7082e+05	1012	17	"Pancakes"
14	5.101e+05	1011	19	"Shawl"
15	8.9975e+05	1011	20	"Snacks"

Close Database Connection

```
close(conn)
```

See Also

mysql | close | sqlread | sqlwrite | rollback

More About

- "Create Table and Add Column Using MySQL Native Interface" on page 6-18
- "Delete Data from Database Using MySQL Native Interface" on page 6-19

Create Table and Add Column Using MySQL Native Interface

This example shows how to connect to a database and manage the database structure using the MySQL® native interface. You can manage the database structure using the `execute` function.

Create Database Connection

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password.

```
datasource = "MySQLDataSource";  
username = "root";  
password = "matlab";  
conn = mysql(datasource,username,password);
```

Create Database Table

Use the SQL `CREATE` statement to create the database table `Person`.

```
sqlquery = strcat("CREATE TABLE Person(lastname VARCHAR(250), ", ...  
    "firstname VARCHAR(250), address VARCHAR(300), age INT)");
```

Create the table in the database using the database connection.

```
execute(conn,sqlquery)
```

Add Database Column

Use the SQL `ALTER` statement to add the database column `city` to the table `Person`.

```
sqlquery = "ALTER TABLE Person ADD city VARCHAR(30)";  
execute(conn,sqlquery)
```

Close Database Connection

```
close(conn)
```

See Also

`mysql` | `close` | `execute`

More About

- “Delete Data from Database Using MySQL Native Interface” on page 6-19
- “Roll Back Data in Database Using MySQL Native Interface” on page 6-16

External Websites

- [MySQL Documentation](#)

Delete Data from Database Using MySQL Native Interface

This example shows how to delete data from a database using MATLAB®. Create the SQL statement using deletion SQL syntax; consult your database documentation for the correct syntax. Execute the delete operation on your database using the `execute` function with the SQL statement. This example demonstrates deleting records from a MySQL® database.

Create Database Connection

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password.

```
datasource = "MySQLDataSource";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

The SQL query `sqlquery` selects all rows of data in the table `inventoryTable`. Execute this SQL query using the database connection. Import the data from the executed query using the `fetch` function, and display the last few rows.

```
sqlquery = "SELECT * FROM inventoryTable";
data = fetch(conn,sqlquery);
tail(data,3)
```

```
ans=3x4 table
   productNumber   Quantity   Price   inventoryDate
   _____   _____   _____   _____
           11           567           0   "2012-09-11 00:30:24"
           12          1278           0   "2010-10-29 18:17:47"
           13          1700          14.5   "2009-05-24 10:58:59"
```

Delete Specific Record

Delete the record for the product number 13 from the table `inventoryTable`. Specify the product number using the `WHERE` clause in the SQL statement `sqlquery`.

```
sqlquery = "DELETE FROM inventoryTable WHERE productnumber = 13";
execute(conn,sqlquery)
```

Display the data in the table `inventorytable` after the deletion. The record with product number 13 is missing.

```
sqlquery = "SELECT * FROM inventoryTable";
data = fetch(conn,sqlquery);
tail(data,3)
```

```
ans=3x4 table
   productNumber   Quantity   Price   inventoryDate
   _____   _____   _____   _____
           10           723           24   "2012-03-14 13:13:09"
           11           567           0   "2012-09-11 00:30:24"
           12          1278           0   "2010-10-29 18:17:47"
```

Close Database Connection

`close(conn)`

See Also

`mysql` | `close` | `fetch` | `execute`

More About

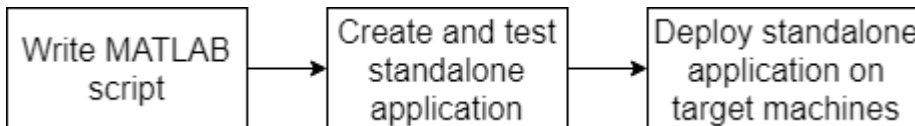
- “Create Table and Add Column Using MySQL Native Interface” on page 6-18
- “Roll Back Data in Database Using MySQL Native Interface” on page 6-16

External Websites

- [MySQL Documentation](#)

Deploy MySQL Native Interface Database Application with MATLAB Compiler

This example shows how to write a script to analyze data stored in a MySQL® database using the MySQL native interface, and deploy the script as a standalone application. Write code that connects to a database, imports data from the database into MATLAB®, analyzes the data, and closes the database connection. Then, you can deploy the code by compiling it as a standalone application using the Application Compiler (MATLAB Compiler) app, and then running the application on other machines.



The example uses the MySQL native interface to create the database connection. Overall, the example follows the steps described in “Create Standalone Application from MATLAB Function” (MATLAB Compiler) and updates the steps for a standalone database application.

Before you begin, note the following:

- You must first install the MySQL/C++ Connector file on each machine where you plan to run the standalone application. For details about configuring the data source, see “Configure MySQL Native Interface Data Source” on page 6-2.
- You must have administrator privileges on each machine where you plan to run the standalone application.

Create Function in MATLAB

Create a MATLAB script named `importMySQLNative.m` and save it in a file location of your choice. The script contains the `importMySQLNative` function, which returns the maximum product number from the data in the `productTable` database table. The function connects to a MySQL database and imports all data from `productTable`. Then, the function calculates the maximum product number.

You must use the syntax with name-value pair arguments when you connect to the database using the `mysql` function.

type `importMySQLNative.m`

```

function maxProdNum = importMySQLNative
% IMPORTMYSQLNATIVE The importMySQLNative function connects to a MySQL®
% database using the MySQL native interface, imports data from the
% database into MATLAB®, performs a simple data analysis, and closes the
% database connection. The database contains a table named |productTable|.

%%
% Connect to the database by using name-value pair arguments of the |mysql|
% function to specify a connection to a MySQL database. For example, this
% code assumes that you are using the user name |username|, password |pwd|,
% database |dbname|, database server |sname|, and port number |3306|.
conn = mysql("username","pwd", ...
            "DatabaseName","dbname", ...
            "Server","sname", ...
  
```

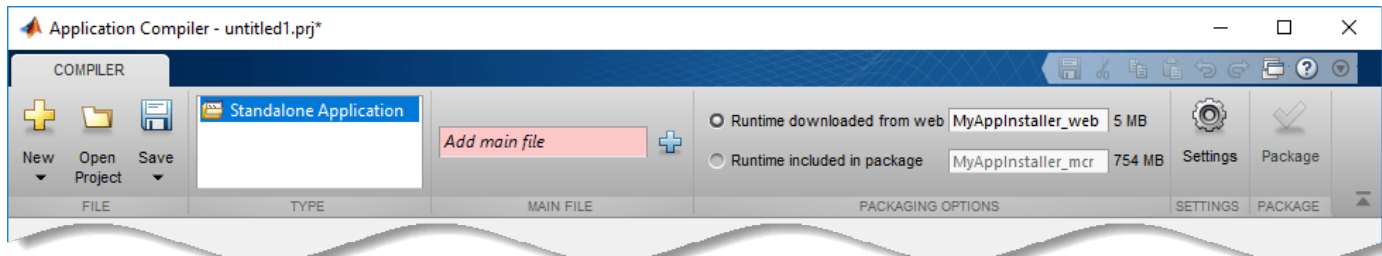
```

        "PortNumber",3306);
%%
% Import data from the |productTable| database table.
tablename = "productTable";
data = sqlread(conn,tablename);
%%
% Determine the highest product number among products.
prodNums = data.productnumber;
maxProdNum = max(prodNums);
%%
% Close the database connection.
close(conn)
end


```

Create Standalone Application Using Application Compiler App

On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow to open the apps gallery. Under **Application Deployment**, click **Application Compiler**.



In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.

- 1 In the **Main File** section of the toolbar, click the plus sign button  to add a main file.
- 2 In the **Add Files** dialog box, browse to the file location that contains your saved script. Select `importMySQLNative.m` and click **Open**. The Application Compiler app adds the `importMySQLNative` function to the list of main files.

Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:

- **Runtime downloaded from web** — Generates an installer that downloads MATLAB Runtime and installs it along with the deployed MATLAB application
- **Runtime included in package** — Generates an installer that includes the MATLAB Runtime installer

Customize the packaged application and its appearance by specifying the following options:

The screenshot displays the MATLAB Compiler application configuration window. The 'Application information' section is active, showing the following fields: 'Application Name' (with a version of 1.0), 'Author Name', 'Email', 'Company', 'Summary', and 'Description'. A 'Set as default contact' button is located next to the 'Company' field. To the right, a splash screen preview is shown with a 'Select custom splash screen' button. Below the 'Application information' section are four expandable sections: 'Additional installer options', 'Files required for your application to run', 'Files installed for your end user', and 'Additional runtime settings', each with a plus sign button for expansion.

- **Application information** — Editable information about the deployed application. You can also customize the appearance of the standalone application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata.
- **Additional installer options** — Options for editing the default installation path for the generated installer and selecting a custom logo.
- **Files required for your application to run** — Additional files required by the generated application to run. The software includes these files in the generated application installer.
- **Files installed for your end user** — Files that are installed with your application. These files include the generated `readme.txt` file and the generated executable for the target platform. To create a database connection using the standalone application, add the MySQL/C++ Connector file.
- **Additional runtime settings** — Platform-specific options for controlling the generated executable.

For details about these options, see “Customize an Application” (MATLAB Compiler).

To generate the packaged application, click **Package** in the **Package** section on the toolbar. In the Save Project dialog box, specify the location in which to save the project.

In the **Package** dialog box, verify that the option **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output:

- `for_redistribution` — Folder containing the file that installs the application and MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc` (such as binary, JAR, header, and source files for a specific target). Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler™.

Install and Run Standalone Application

To install the standalone application, double-click the `MyAppInstaller_web` executable in the `for_redistribution` folder.

If you want to connect to the Internet using a proxy server, click **Connection Settings**. Enter the proxy server settings in the provided dialog box. Click **OK**.

To complete the installation, follow the instructions in the installation wizard.

To run your standalone application:

- 1 Open a terminal window.
- 2 Navigate to the folder in which you installed the application.
- 3 Run the application.

Test Standalone Application on Target Machine

Choose one target machine to test the MATLAB generated standalone application.

Copy the files in the `for_testing` folder to the target machine.

To test your standalone application:

- 1 Open a terminal window.
- 2 Navigate to the `for_testing` folder.
- 3 Run the application.

Deploy Standalone Application on Target Machines

Copy the `for_redistribution_files_only` folder to a file location on all target machines where MATLAB or MATLAB Runtime is installed.

Run the MATLAB generated standalone application on all target machines by using the executable in the `for_redistribution_files_only` folder.

See Also

`mysql` | `close` | `sqlread`

More About

- “Create Functions in Files”
- “Create Standalone Application from MATLAB Function” (MATLAB Compiler)
- “Customize an Application” (MATLAB Compiler)
- “Import Data from MySQL Database Table” on page 6-6

External Websites

- [MySQL Documentation](#)

PostgreSQL Native Interface Topics

Configure PostgreSQL Native Interface Data Source

This tutorial shows how to set up a PostgreSQL native interface data source and connect to a PostgreSQL database using the Database Explorer app or the command line. The tutorial uses the libpq driver version 10.12 to connect to a PostgreSQL version 9.405 database.

The configuration steps show configuring the PostgreSQL native data source on the Windows platform. The same steps work on Linux and macOS platforms.

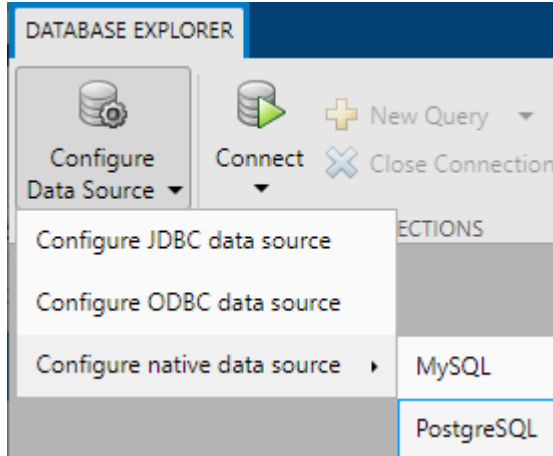
Database Toolbox includes the libpq driver. Therefore, you do not have to install and configure the driver.

Step 1. Set up the data source.

You can set up a data source using the Database Explorer app or the command line.

Set Up Data Source Using Database Explorer App

- 1 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**. Alternatively, enter `databaseExplorer` at the command line.
- 2 In the **Data Source** section, select **Configure Data Source > Configure native data source > PostgreSQL**.



The PostgreSQL Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter the name of your data source. (This example uses a data source named `PostgreSQLDataSource`.) You use this name to establish a connection to your database.

PostgreSQL Data Source Configuration

Data Source Details

Name: PostgreSQLDataSource

Connection Parameters

Database Name:

Server: localhost

Port Number: 5432

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

- 4 In the **Database Name** box, enter the name of your database. In the **Server** box, enter the name of your database server. Consult your database administrator for the name of your database server. In the **Port Number** box, enter the port number.
- 5 Under **Connection Options**, in the **Name** column, enter the name of an additional driver-specific option. Then, in the **Value** column, enter the value of the driver-specific option. Click the plus sign + to specify additional driver-specific options.
- 6 Click **Test**. The Test Connection dialog box opens. Enter the user name and password for your database. Click **Test**.

If your connection succeeds, the Database Explorer dialog box displays a message indicating the connection is successful. Otherwise, it displays an error message.

- 7 Click **Save**. The PostgreSQL Data Source Configuration dialog box displays a message indicating the data source is saved successfully. Close this dialog box.

Set Up Data Source Using Command Line

- 1 Create a PostgreSQL native interface data source for a PostgreSQL database.

```
vendor = "PostgreSQL";  
opts = databaseConnectionOptions("native", vendor);
```

- 2 Set the database connection options. For example, this code assumes that you are connecting to a data source named PostgreSQLDataSource, database name toystore_doc, database server dbtb00, and port number 5432.

```
opts = setoptions(opts, ...  
    'DataSourceName', "PostgreSQLDataSource", ...  
    'DatabaseName', "toystore_doc", 'Server', "dbtb00", ...  
    'PortNumber', 5432);
```

- 3 Test the database connection by specifying the user name and password.

```
username = "dbdev";  
password = "matlab";  
status = testConnection(opts, username, password);
```

- 4 Save the data source.

```
saveAsDataSource(opts)
```

After you complete the data source setup, connect to the PostgreSQL database using the Database Explorer app or the command line.

Step 2. Connect using the Database Explorer app or the command line.

Connect to PostgreSQL Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Connections** section, click **Connect** and select the data source for the connection.
- 2 In the connection dialog box, enter a user name and password. Click **Connect**.

The Catalog and Schema dialog box opens.

- 3 Select the catalog and schema from the **Catalog** and **Schema** lists. Click **OK**.

The app connects to the database and displays its tables in the **Data Browser** pane. A data source tab appears to the right of the pane. The title of the data source tab is the data source name that you defined during the setup. The data source tab contains empty **SQL Query** and **Data Preview** panes.

- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the data source tab to close the SQL query. In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Connect to PostgreSQL Using Command Line

- 1 Connect to a PostgreSQL database using the configured PostgreSQL native interface data source, user name, and password.

```
datasource = "PostgreSQLDataSource";  
username = "dbdev";  
password = "matlab";  
conn = postgresql(datasource,username,password);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

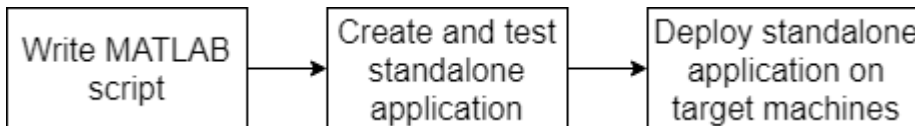
databaseConnectionOptions | saveAsDataSource | setoptions | testConnection |
postgresql | close

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-10
- “Data Preview Using Database Explorer App” on page 4-14
- “Modify and Delete Data Sources” on page 4-17
- “Database Explorer App Error Messages” on page 3-13

Deploy PostgreSQL Native Interface Database Application with MATLAB Compiler

This example shows how to write a script to analyze data stored in a PostgreSQL database using the PostgreSQL native interface, and deploy the script as a standalone application. Write code that connects to a database using the PostgreSQL native interface, imports data from the database into MATLAB®, analyzes the data, and closes the database connection. Then, you can deploy the code by compiling it as a standalone application using the Application Compiler (MATLAB Compiler) app, and then running the application on other machines.



Overall, the example follows the steps described in “Create Standalone Application from MATLAB Function” (MATLAB Compiler) and updates the steps for a standalone database application.

You must have administrator privileges on each machine where you plan to run the standalone application.

Create Function in MATLAB

Create a MATLAB script named `importPostgreSQLNative.m` and save it in a file location of your choice. The script contains the `importPostgreSQLNative` function, which returns the maximum product number from the data in the `productTable` database table. The function connects to a PostgreSQL database and imports all data from `productTable`. Then, the function calculates the maximum product number.

You must use the syntax with name-value pair arguments when you connect to the database using the `postgresql` function.

type `importPostgreSQLNative.m`

```

function maxProdNum = importPostgreSQLNative
% IMPORTPOSTGRESQLNATIVE The importPostgreSQLNative function connects to a
% PostgreSQL database using the PostgreSQL native interface, imports data
% from the database into MATLAB®, performs a simple data analysis, and
% closes the database connection. The database contains a table named
% |productTable|.
%%
% Connect to the database by using name-value pair arguments of the
% |postgresql| function to specify a connection to a PostgreSQL database.
% For example, this code assumes that you are using the user name
% |username|, password |pwd|, database |dbname|, database server |sname|,
% and port number |5432|.
conn = postgresql("username","pwd", ...
    "DatabaseName","dbname", ...
    "Server","sname", ...
    "PortNumber",5432);
%%
% Import data from the |productTable| database table.
tablename = "productTable";
data = sqlread(conn,tablename);
  
```

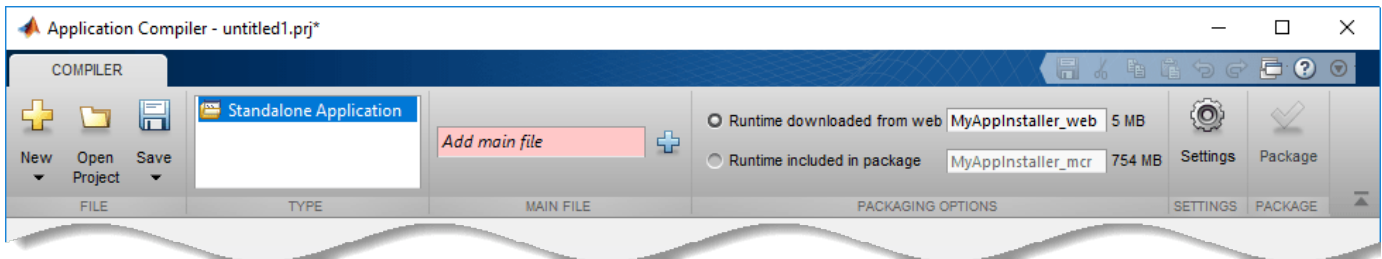
```

%%
% Determine the highest product number among products.
prodNums = data.productnumber;
maxProdNum = max(prodNums);
%%
% Close the database connection.
close(conn)
end


```

Create Standalone Application Using Application Compiler App

On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow to open the apps gallery. Under **Application Deployment**, click **Application Compiler**.



In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.

- 1 In the **Main File** section of the toolbar, click the plus sign button  to add a main file.
- 2 In the **Add Files** dialog box, browse to the file location that contains your saved script. Select `importPostgreSQLNative.m` and click **Open**. The Application Compiler app adds the `importPostgreSQLNative` function to the list of main files.

Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:

- **Runtime downloaded from web** — Generates an installer that downloads MATLAB Runtime and installs it along with the deployed MATLAB application
- **Runtime included in package** — Generates an installer that includes the MATLAB Runtime installer

Customize the packaged application and its appearance by specifying the following options:

The screenshot shows the 'Application information' dialog box. It features a header bar with the title 'Application information'. Below the header, there is a grid of input fields: 'Application Name' (with a version field set to '1.0'), 'Author Name', 'Email', 'Company', 'Summary', and 'Description'. A 'Set as default contact' button is located to the right of the 'Company' field. To the right of the main form is a preview of a splash screen with a 'Select custom splash screen' button. Below the main form are three expandable sections: 'Additional installer options', 'Files required for your application to run', and 'Files installed for your end user', each with a plus sign button. At the bottom is 'Additional runtime settings'.

- **Application information** — Editable information about the deployed application. You can also customize the appearance of the standalone application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata.
- **Additional installer options** — Options for editing the default installation path for the generated installer and selecting a custom logo.
- **Files required for your application to run** — Additional files required by the generated application to run. The software includes these files in the generated application installer.
- **Files installed for your end user** — Files that are installed with your application. These files include the generated `readme.txt` file and the generated executable for the target platform.
- **Additional runtime settings** — Platform-specific options for controlling the generated executable.

For details about these options, see “Customize an Application” (MATLAB Compiler).

To generate the packaged application, click **Package** in the **Package** section on the toolbar. In the Save Project dialog box, specify the location in which to save the project.

In the **Package** dialog box, verify that the option **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output:

- `for_redistribution` — Folder containing the file that installs the application and MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc` (such as binary, JAR, header, and source files for a specific target). Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler™.

Install and Run Standalone Application

To install the standalone application, double-click the `MyAppInstaller_web` executable in the `for_redistribution` folder.

If you want to connect to the Internet using a proxy server, click **Connection Settings**. Enter the proxy server settings in the provided dialog box. Click **OK**.

To complete the installation, follow the instructions in the installation wizard.

To run your standalone application:

- 1 Open a terminal window.
- 2 Navigate to the folder in which you installed the application.
- 3 Run the application.

Test Standalone Application on Target Machine

Choose one target machine to test the MATLAB generated standalone application.

Copy the files in the `for_testing` folder to the target machine.

To test your standalone application:

- 1 Open a terminal window.
- 2 Navigate to the `for_testing` folder.
- 3 Run the application.

Deploy Standalone Application on Target Machines

Copy the `for_redistribution_files_only` folder to a file location on all target machines where MATLAB or MATLAB Runtime is installed.

Run the MATLAB generated standalone application on all target machines by using the executable in the `for_redistribution_files_only` folder.

See Also

`postgresql` | `close` | `sqlread`

More About

- “Create Functions in Files”
- “Create Standalone Application from MATLAB Function” (MATLAB Compiler)
- “Customize an Application” (MATLAB Compiler)
- “Import Data from PostgreSQL Database Table” on page 7-11

Import Data from PostgreSQL Database Table

This example shows how to import data from a table in a PostgreSQL database into the MATLAB® workspace using the `sqlread` and `fetch` functions with the PostgreSQL native interface.

Connect to Database

Create a PostgreSQL native interface database connection using the data source name `PostgreSQLDataSource` and a user name and password. The PostgreSQL database contains the table `productTable`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Import Data from Database Table Using `sqlread` Function

Import product data from the database table `productTable` by using the `sqlread` function and the database connection. This function imports data as a MATLAB table.

```
tablename = "productTable";
data = sqlread(conn,tablename);
```

Display the product number and description in the imported data.

```
data(:, [1 5])
```

```
ans=11x2 table
   productnumber  productdescription
   _____  _____
           9      "Victorian Doll"
           8      "Train Set"
           7      "Engine Kit"
           2      "Painting Set"
           4      "Space Cruiser"
           1      "Building Blocks"
           5      "Tin Soldier"
           6      "Sail Boat"
           3      "Slinky"
          10      "Teddy Bear"
          30      "Rubik's Cube"
```

Import Data from Database Table Using `fetch` Function

Import product data from the database table `productTable` by using the `fetch` function and the database connection. Create an SQL query to import data that is sorted by product description alphabetically. The `fetch` function imports data as a MATLAB table.

```
sqlquery = "SELECT * FROM productTable ORDER BY productDescription ASC";
data = fetch(conn,sqlquery);
```

Display the product number and description in the imported data.

```
data(:, [1 5])
```

```
ans=11x2 table
  productnumber  productdescription
  _____  _____
         1      "Building Blocks"
         7      "Engine Kit"
         2      "Painting Set"
        30      "Rubik's Cube"
         6      "Sail Boat"
         3      "Slinky"
         4      "Space Cruiser"
        10      "Teddy Bear"
         5      "Tin Soldier"
         8      "Train Set"
         9      "Victorian Doll"
```

Close Database Connection

```
close(conn)
```

See Also

[postgresql](#) | [close](#) | [sqlread](#) | [fetch](#)

More About

- “Customize Options for Importing Data from PostgreSQL Database into MATLAB” on page 7-13

External Websites

- [PostgreSQL Documentation](#)

Customize Options for Importing Data from PostgreSQL Database into MATLAB

This example shows how to customize import options when importing data from a database table using the PostgreSQL native interface. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options for multiple database columns. Import data using the `sqlread` function.

The example uses the `patients.xls` spreadsheet, which contains patient information. Also, the example uses a PostgreSQL database version 9.405 database and the `libpq` driver version 10.12.

Create Database Connection

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Load Example Data

Load patient information into the MATLAB® workspace.

```
patients = readtable("patients.xls");
```

Create the `patients` database table using the patient information.

```
tablename = "patients";
sqlwrite(conn,tablename,patients)
```

Create SQLImportOptions Object

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename);
```

Display the default data types of the variables by accessing the `VariableNames` and `VariableTypes` properties of the `SQLImportOptions` object using dot notation.

```
disp([opts.VariableNames' opts.VariableTypes'])

    {'lastname'          }    {'string'  }
    {'gender'            }    {'string'  }
    {'age'               }    {'double'  }
    {'location'         }    {'string'  }
    {'height'           }    {'double'  }
    {'weight'           }    {'double'  }
    {'smoker'           }    {'logical' }
    {'systolic'         }    {'double'  }
    {'diastolic'        }    {'double'  }
    {'selfassessedhealthstatus'} {'string'  }
```

Customize Import Options

Change the data types of multiple variables. Convert the data type for all text variables to char. Also, convert the data type for all numeric variables to single.

```
textvars = ["lastname" "gender" "location" "selfassessedhealthstatus"];
opts = setoptions(opts,textvars,'Type',"char");

numvars = ["age" "height" "weight" "systolic" "smoker" "diastolic"];
opts = setoptions(opts,numvars,'Type',"single");
```

Display the updated data types of the variables.

```
disp([opts.VariableNames' opts.VariableTypes'])

{'lastname'      }      {'char'  }
{'gender'        }      {'char'  }
{'age'           }      {'single'}
{'location'      }      {'char'  }
{'height'        }      {'single'}
{'weight'        }      {'single'}
{'smoker'        }      {'single'}
{'systolic'      }      {'single'}
{'diastolic'     }      {'single'}
{'selfassessedhealthstatus'} {'char'  }
```

Set the import options to replace missing data in the specified variables with the fill value unknown.

```
varnames = ["lastname" "location"];
opts = setoptions(opts,varnames,'FillValue',"unknown");
```

Set the import options to omit rows with missing data in the lastname variable.

```
varname = "lastname";
opts = setoptions(opts,varname,'MissingRule',"omitrow");
```

Preview Data Before Importing

Before importing the data, preview it by using the customized import options.

```
T = preview(opts)
```

```
T=8x10 table
  lastname      gender      age      location      height      weight      sm
```

lastname	gender	age	location	height	weight	sm
{'Smith' }	{'Male' }	38	{'County General Hospital' }	71	176	:
{'Johnson' }	{'Male' }	43	{'VA Hospital' }	69	163	(
{'Williams' }	{'Female' }	38	{'St. Mary's Medical Center' }	64	131	(
{'Jones' }	{'Female' }	40	{'VA Hospital' }	67	133	(
{'Brown' }	{'Female' }	49	{'County General Hospital' }	64	119	(
{'Davis' }	{'Female' }	46	{'St. Mary's Medical Center' }	68	142	(
{'Miller' }	{'Female' }	33	{'VA Hospital' }	64	142	:
{'Wilson' }	{'Male' }	40	{'VA Hospital' }	68	180	(

Import Data Using Import Options

Import the variables with the customized data types by using the `sqlread` function, and display the first eight rows of imported data.

```
T = sqlread(conn,tablename,opts);
head(T)
```

```
ans=8x10 table
      lastname      gender      age      location      height      weight      sm
      _____      _____      _____      _____      _____      _____      _____
      {'Smith' }      {'Male' }      38      {'County General Hospital' }      71      176      :
      {'Johnson' }      {'Male' }      43      {'VA Hospital' }      69      163      (
      {'Williams' }      {'Female' }      38      {'St. Mary's Medical Center' }      64      131      (
      {'Jones' }      {'Female' }      40      {'VA Hospital' }      67      133      (
      {'Brown' }      {'Female' }      49      {'County General Hospital' }      64      119      (
      {'Davis' }      {'Female' }      46      {'St. Mary's Medical Center' }      68      142      (
      {'Miller' }      {'Female' }      33      {'VA Hospital' }      64      142      :
      {'Wilson' }      {'Male' }      40      {'VA Hospital' }      68      180      (
```

Delete Example Data and Close Database Connection

Delete the patients database table using the `execute` function.

```
sqlquery = strcat("DROP TABLE ",tablename);
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

See Also

Objects

SQLImportOptions

Functions

postgresql | close | sqlread | databaseImportOptions | preview | setoptions | sqlwrite | execute

More About

- “Import Data from PostgreSQL Database Table” on page 7-11

External Websites

- PostgreSQL Documentation

Import Large PostgreSQL Data Using DatabaseDatastore Object

This example shows how to use the `databaseDatastore` function to create a `DatabaseDatastore` object for accessing collections of data stored in a PostgreSQL database. After creating a `DatabaseDatastore` object, you can preview data, read data in chunks, and read every record in the data set.

To analyze large data, you can run algorithms on large data sets using a tall array. Alternatively, you can write a MapReduce algorithm that defines the chunking and reduction of the data.

This example uses a preconfigured PostgreSQL data source to create the database connection. For more information, see the `databaseConnectionOptions` function.

Create DatabaseDatastore Object

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Create a `DatabaseDatastore` object using the database connection and an SQL query. This query retrieves all data from the `airlinesmall` table.

```
sqlquery = "select * from airlinesmall";
dbds = databaseDatastore(conn,sqlquery);
```

Preview Data in DatabaseDatastore Object

Preview the first eight records in the data set returned by executing the SQL query.

```
preview(dbds)
```

```
ans=8x29 table
```

year	month	dayofmonth	dayofweek	deptime	crsdeptime	arrtime	crsarrrtime
1990	9	11	2	1810	1812	1939	1930
1990	10	27	6	1353	1355	1634	1640
1990	10	23	2	1057	1055	1205	1155
1990	10	8	1	1515	1440	1609	1535
1990	10	19	5	1130	1120	1203	1154
1990	10	12	5	1755	1733	1858	1820
2001	11	22	4	1345	1355	1530	1549
2001	11	26	1	2105	2110	2209	2237

Read Data in DatabaseDatastore Object

Read the data and display the first few records.

```
data = read(dbds);
head(data)
```



```
ans=8x29 table
```

year	month	dayofmonth	dayofweek	deptime	crsdeptime	arrtime	crsarrrtime
1990	9	11	2	1810	1812	1939	1930
1990	10	27	6	1353	1355	1634	1640
1990	10	23	2	1057	1055	1205	1155
1990	10	8	1	1515	1440	1609	1535
1990	10	19	5	1130	1120	1203	1154
1990	10	12	5	1755	1733	1858	1820
2001	11	22	4	1345	1355	1530	1549
2001	11	26	1	2105	2110	2209	2237

Read the DatabaseDatastore object two more times by using the counter n. Read 10 records at a time.

```
n = 0;
while(hasdata(dbds) && n~=2)
    read(dbds)
    n = n+1;
end
```

```
ans=10000x29 table
```

year	month	dayofmonth	dayofweek	deptime	crsdeptime	arrtime	crsarrrtime
1997	12	10	3	1812	1730	2149	2047
1997	12	17	3	1806	1800	1932	1933
1997	12	27	6	1803	1805	1947	1945
1997	12	27	6	807	800	1138	1130
1997	12	20	6	1501	1120	1828	1432
1998	1	18	7	1207	1211	1358	1355
1998	1	22	4	2022	0	2151	0
1998	1	6	2	730	730	840	840
1998	1	5	1	730	730	1040	1100
1998	1	16	5	1415	1415	1623	1615
1998	1	28	3	1017	1000	1222	1210
1998	1	7	3	2018	2020	2231	2242
1998	1	2	5	915	915	1113	1115
1998	1	4	7	1035	1030	1332	1356
1998	1	16	5	1235	1210	1436	1415
1998	1	13	2	802	800	1047	1053
:							

```
ans=10000x29 table
```

year	month	dayofmonth	dayofweek	deptime	crsdeptime	arrtime	crsarrrtime
1988	1	5	2	2149	2045	2333	2232
1988	1	28	4	1005	1000	1137	1131
1988	1	13	3	1159	1200	1514	1518
1988	1	25	1	630	630	948	944
1988	1	12	2	2005	1900	2131	2036
1988	1	3	7	1153	1155	1202	1205
1988	1	8	5	1227	1200	1413	1330
1988	1	29	5	1456	1500	1520	1526

1988	1	27	3	37	40	107	115
1988	1	25	1	638	640	1119	1039
1988	1	11	1	1313	1130	1615	1434
1988	1	19	2	901	842	1051	1036
1988	1	12	2	1737	1730	1953	1947
1988	1	12	2	1122	1125	1357	1407
1988	1	13	3	1728	1720	1957	1942
1988	1	12	2	1559	1500	1844	1745
:							

Reset DatabaseDatastore Object

Reset the DatabaseDatastore object to its original state, where no data has been read from it. Resetting allows you to reread from the same DatabaseDatastore object.

```
reset(dbds)
```

Read Every Record in DatabaseDatastore Object

Read every record in the DatabaseDatastore object.

```
data = readall(dbds);
```

Display the first three records of the full data set.

```
head(data,3)
```

```
ans=3x29 table
```

year	month	dayofmonth	dayofweek	deptime	crsdeptime	arrtime	crsarrrtime
1990	9	11	2	1810	1812	1939	1930
1990	10	27	6	1353	1355	1634	1640
1990	10	23	2	1057	1055	1205	1155

Close DatabaseDatastore Object and Database Connection

```
close(dbds)
```

See Also

[postgresql](#) | [databaseDatastore](#) | [preview](#) | [read](#) | [readall](#) | [reset](#) | [close](#)

More About

- “Import Data from PostgreSQL Database Table” on page 7-11

External Websites

- [PostgreSQL Documentation](#)

Insert Data into Database Table Using PostgreSQL Native Interface

This example shows how to import data from a database into MATLAB®, perform calculations on the data, and export the results to a database table.

The example assumes that you are connecting to a PostgreSQL database that contains tables named `salesvolume` and `yearlysales`. The `salesvolume` table contains the column names for each month. The `yearlysales` table contains the column names `month` and `salestotal`. Also, the example uses a PostgreSQL database version 9.4.05 database and the libpq driver version 10.12.

Create Database Connection

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Calculate Sum of Sales Volume for One Month

Import sales volume data for the month of March using the database connection. The `salesvolume` database table contains sales volume data.

```
tablename = "salesvolume";
data = sqlread(conn,tablename);
```

Display the first three rows of sales volume data. The fourth variable contains the data for the month of March.

```
head(data(:,4),3)
```

```
ans=3x1 table
    march
-----
    981
   1414
    890
```

Calculate the sum of the March sales. Assign the result to the MATLAB workspace variable `total` and display the result.

```
total = sum(data.march)
```

```
total = 14606
```

Insert Total Sales for One Month into Database

Retrieve the name of the month from the sales volume data.

```
month = data.Properties.VariableNames(4);
```

Define the names of the columns for the data to insert as a string array.

```
colnames = ["month" "salestotal"];
```

Create a MATLAB table that stores the data to export.

```
results = table(month,total,'VariableNames',colnames);
```

Determine the status of the `AutoCommit` database flag. This status determines whether or not the insert action can be undone.

```
conn.AutoCommit
```

```
ans =  
"on"
```

The `AutoCommit` flag is set to `on`. The database commits the exported data automatically to the database, and this action cannot be undone.

Insert the sum of sales for the month of March into the `yearlysales` table.

```
tablename = "yearlysales";  
sqlwrite(conn,tablename,results)
```

Import the data from the `yearlysales` table. This data contains the calculated result.

```
data = sqlread(conn,tablename)
```

```
data=1x2 table  
    month    salestotal  
-----  
"march"    14606
```

Close Database Connection

```
close(conn)
```

See Also

`postgresql` | `close` | `sqlread` | `sqlwrite`

More About

- “Import Data from PostgreSQL Database Table” on page 7-11

Roll Back Data in Database Using PostgreSQL Native Interface

This example shows how to connect to a database, insert a row into an existing database table, and then roll back (or reverse) the insert using the PostgreSQL native interface. The example uses a PostgreSQL database version 9.4.05 database and the libpq driver version 10.12. The database contains the table `productTable`.

Create Database Connection

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Append Data to Existing Database Table

Set the `AutoCommit` property of the connection object to `off`. Any updates you make after turning off this flag are not committed to the database automatically.

```
conn.AutoCommit = "off";
```

To view the existing database table `productTable` before appending data, import its contents into MATLAB® and display the last few rows.

```
tablename = "productTable";
rows = sqlread(conn,tablename);
tail(rows,3)
```

```
ans=3x5 table
   productnumber   stocknumber   suppliernumber   unitcost   productdescription
   _____   _____   _____   _____   _____
           6       4.0088e+05       1004           8       "Sail Boat"
           3       4.01e+05       1009          17       "Slinky"
          10       8.8865e+05       1006          24       "Teddy Bear"
```

Create a MATLAB table that contains the data for one product.

```
data = table(30,500000,1000,25,"Rubik's Cube", ...
    'VariableNames',["productnumber" "stocknumber" ...
    "suppliernumber" "unitcost" "productdescription"]);
```

Append the product data into the database table `productTable`.

```
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB again, and display the last few rows. The results contain a new row for the inserted product.

```
rows = sqlread(conn,tablename);
tail(rows,4)
```

```
ans=4x5 table
  productnumber  stocknumber  suppliernumber  unitcost  productdescription
  _____  _____  _____  _____  _____
           6      4.0088e+05      1004           8      "Sail Boat"
           3      4.01e+05       1009          17      "Slinky"
          10      8.8865e+05      1006          24      "Teddy Bear"
          30           5e+05       1000          25      "Rubik's Cube"
```

Roll Back Data

Roll back the inserted row.

```
rollback(conn)
```

Import the contents of the database table into MATLAB again, and display the last few rows. The results no longer contain the inserted row.

```
rows = sqlread(conn,tablename);
tail(rows,3)
```

```
ans=3x5 table
  productnumber  stocknumber  suppliernumber  unitcost  productdescription
  _____  _____  _____  _____  _____
           6      4.0088e+05      1004           8      "Sail Boat"
           3      4.01e+05       1009          17      "Slinky"
          10      8.8865e+05      1006          24      "Teddy Bear"
```

Close Database Connection

```
close(conn)
```

See Also

postgresql | close | sqlread | sqlwrite | rollback

More About

- “Import Data from PostgreSQL Database Table” on page 7-11
- “Insert Data into Database Table Using PostgreSQL Native Interface” on page 7-19

Create Table and Add Column Using PostgreSQL Native Interface

This example shows how to connect to a database and manage the database structure using the PostgreSQL native interface. You can manage the database structure using the `execute` function.

Create Database Connection

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Create Database Table

Use the SQL `CREATE` statement to create the database table `Person`.

```
sqlquery = strcat("CREATE TABLE Person(lastname varchar,", ...
    "firstname varchar,address varchar,age numeric)");
```

Create the table in the database using the database connection.

```
execute(conn,sqlquery)
```

Add Database Column

Use the SQL `ALTER` statement to add the database column `city` to the table `Person`.

```
sqlquery = "ALTER TABLE Person ADD city varchar(30)";
execute(conn,sqlquery)
```

Close Database Connection

```
close(conn)
```

See Also

[postgresql](#) | [close](#) | [execute](#)

More About

- “Insert Data into Database Table Using PostgreSQL Native Interface” on page 7-19
- “Roll Back Data in Database Using PostgreSQL Native Interface” on page 7-21

External Websites

- [PostgreSQL Documentation](#)

Delete Data from Database Using PostgreSQL Native Interface

This example shows how to delete data from a PostgreSQL database using MATLAB®. First, create an SQL statement with the deletion SQL syntax. Consult your database documentation for the correct syntax. Execute the delete operation on your database using the `execute` function with the SQL statement.

Create Database Connection

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

The SQL query `sqlquery` selects all rows of data in the table `inventorytable`. Execute this SQL query using the database connection. Import the data from the executed query using the `fetch` function, and display the last few rows.

```
sqlquery = "SELECT * FROM inventorytable";
data = fetch(conn,sqlquery);
tail(data,3)
```

```
ans=3x4 table
  productnumber  quantity  price  inventorydate
  _____  _____  _____  _____
           11           567         0  "2012-09-11 00:30:24"
           12          1278         0  "2010-10-29 18:17:47"
           13          1700        14.5  "2009-05-24 10:58:59"
```

Delete Specific Record

Delete the record for the product number 13 from the table `inventorytable`. Specify the product number using the `WHERE` clause in the SQL statement `sqlquery`.

```
sqlquery = "DELETE FROM inventorytable WHERE productnumber = 13";
execute(conn,sqlquery)
```

Display the data in the table `inventorytable` after the deletion. The record with product number 13 is no longer included.

```
sqlquery = "SELECT * FROM inventorytable";
data = fetch(conn,sqlquery);
tail(data,3)
```

```
ans=3x4 table
  productnumber  quantity  price  inventorydate
  _____  _____  _____  _____
           10           723         24  "2012-03-14 13:13:09"
           11           567         0  "2012-09-11 00:30:24"
           12          1278         0  "2010-10-29 18:17:47"
```


Close Database Connection

`close(conn)`

See Also

`postgresql` | `close` | `execute` | `fetch`

More About

- “Insert Data into Database Table Using PostgreSQL Native Interface” on page 7-19
- “Roll Back Data in Database Using PostgreSQL Native Interface” on page 7-21

External Websites

- [PostgreSQL Documentation](#)

Apache Cassandra Database C++ Interface Topics

Convert CQL Data Types to MATLAB Data Types Using Apache Cassandra Database C++ Interface

When you import data into MATLAB using the Apache Cassandra database C++ interface, the `partitionRead` and `executecql` functions convert the Cassandra Query Language (CQL) data types to MATLAB data types. When you export from MATLAB into a Cassandra database, the `upsert` function converts MATLAB data types to CQL data types. This table describes the CQL data types and shows their corresponding MATLAB data types for data import and export.

CQL Data Type	CQL Data Type Description	Data Type of MATLAB Table Variable for Data Import	Data Type of MATLAB Table Variable for Data Export
<code>ascii</code>	US-ASCII character string	<code>string</code>	<code>char</code> , <code>string</code> , or cell array of character vectors
<code>bigint</code>	64-bit signed long integer	<code>int64</code>	<ul style="list-style-type: none"> • <code>double</code> • <code>single</code> • <code>int8</code> • <code>int16</code> • <code>int32</code> • <code>int64</code> • <code>uint8</code> • <code>uint16</code> • <code>uint32</code> • <code>uint64</code> • <code>logical</code>
<code>blob</code>	Arbitrary bytes (no validation)	<code>uint8</code>	cell array of numeric vectors
<code>boolean</code>	<code>true</code> or <code>false</code>	<code>logical</code>	<ul style="list-style-type: none"> • <code>double</code> • <code>single</code> • <code>int8</code> • <code>int16</code> • <code>int32</code> • <code>int64</code> • <code>uint8</code> • <code>uint16</code> • <code>uint32</code> • <code>uint64</code> • <code>logical</code>
<code>counter</code>	Distributed counter value (64-bit long integer)	<code>int64</code>	Not supported by the <code>upsert</code> function

CQL Data Type	CQL Data Type Description	Data Type of MATLAB Table Variable for Data Import	Data Type of MATLAB Table Variable for Data Export
date	Value is a date with no corresponding time value. Cassandra encodes the date as a 32-bit integer representing days since epoch (January 1, 1970).	datetime array without the time component and time zone	datetime array, char, string, or cell array of character vectors
decimal	Variable-precision decimal	string	<ul style="list-style-type: none"> • string • cell array of character vectors • double • single • int8 • int16 • int32 • int64 • uint8 • uint16 • uint32 • uint64 • sym
double	64-bit IEEE [®] -754 floating point	double	<ul style="list-style-type: none"> • double • single • int8 • int16 • int32 • int64 • uint8 • uint16 • uint32 • uint64 • logical

CQL Data Type	CQL Data Type Description	Data Type of MATLAB Table Variable for Data Import	Data Type of MATLAB Table Variable for Data Export
float	32-bit IEEE-754 floating point	single	<ul style="list-style-type: none"> • double • single • int8 • int16 • int32 • int64 • uint8 • uint16 • uint32 • uint64 • logical
inet	IP address string in IPv4 or IPv6 format	string	char, string, or cell array of character vectors
int	32-bit signed integer	int32	<ul style="list-style-type: none"> • double • single • int8 • int16 • int32 • int64 • uint8 • uint16 • uint32 • uint64 • logical
list<type>	Collection of one or more ordered elements (for example, [literal, literal, literal])	<p>Array of data types, one for each item in the collection.</p> <p>For example, if a Cassandra database table column has the list<int> data type, then each row in the MATLAB table contains an array of int32 data types. In this case, the data type of the MATLAB table variable is a cell array of arrays.</p>	<p>Cell array of vectors, where each vector is compatible with the type of the list.</p> <p>For example, if the Cassandra database table contains list<int>, then the MATLAB table must contain a cell array of numeric vectors.</p>

CQL Data Type	CQL Data Type Description	Data Type of MATLAB Table Variable for Data Import	Data Type of MATLAB Table Variable for Data Export
map<key Type, valueType>	JSON-style array of literals (for example, { literal : literal, literal : literal, ...})	<p>An n-by-2 MATLAB table where n is the number of key-value pairs in the map. The first variable <code>Keys</code> has the keys of the map. The data type for this variable depends on the key type defined for the map. Similarly, the <code>Values</code> variable has the values that correspond to each key. The data type for the <code>Values</code> variable depends on the value type defined for the map.</p> <p>For example, if a Cassandra database table column has the map<text, double> data type, then the <code>partitionRead</code> and <code>executeCQL</code> functions convert this data type into a MATLAB table. The table has the <code>Keys</code> variable as string scalars and the <code>Values</code> variable as a double array. In this case, the data type of the MATLAB table variable is a cell array of tables.</p>	<p>Cell array of tables, where each table contains the <code>Keys</code> and <code>Values</code> variables.</p> <p>The data types of these two variables are compatible with the data types of the keys and values of the Cassandra database map.</p> <p>For example, if the Cassandra database table contains map<int, text>, then the MATLAB table must contain a cell array of tables, where each table has a <code>Keys</code> variable that is numeric and a <code>Values</code> variable that is a string.</p>
set<type>	Collection of one or more elements (for example, {literal, literal, literal})	<p>Array of data types, one for each item in the collection.</p> <p>For example, if a Cassandra database column has the set<float> data type, then each row in the resulting MATLAB table contains an array of single values. In this case, the data type of the MATLAB table variable is a cell array of arrays.</p>	<p>Cell array of vectors, where each vector is compatible with the type of the collection.</p> <p>For example, if the Cassandra database table contains set<int>, then the MATLAB table must contain a cell array of numeric vectors.</p>

CQL Data Type	CQL Data Type Description	Data Type of MATLAB Table Variable for Data Import	Data Type of MATLAB Table Variable for Data Export
<code>smallint</code>	2-byte integer	<code>int16</code>	<ul style="list-style-type: none"> • <code>double</code> • <code>single</code> • <code>int8</code> • <code>int16</code> • <code>int32</code> • <code>int64</code> • <code>uint8</code> • <code>uint16</code> • <code>uint32</code> • <code>uint64</code> • <code>logical</code>
<code>text</code>	UTF-8 encoded string	<code>string</code>	<code>char</code> , <code>string</code> , or cell array of character vectors
<code>time</code>	Cassandra database encodes this value as a 64-bit signed integer that represents the number of nanoseconds since midnight.	<code>duration array</code>	<code>duration array</code> , <code>char</code> , <code>string</code> , or cell array of character vectors
<code>timestamp</code>	Date and time with millisecond precision, encoded as 8 bytes since epoch (January 1, 1970)	<code>datetime array</code> with the date component and time zone as UTC or GMT	<code>datetime array</code> , <code>char</code> , <code>string</code> , or cell array of character vectors
<code>timeuuid</code>	Version 1 UUID only	<code>string</code>	<code>char</code> , <code>string</code> , or cell array of character vectors
<code>tinyint</code>	1-byte integer	<code>int8</code>	<ul style="list-style-type: none"> • <code>double</code> • <code>single</code> • <code>int8</code> • <code>int16</code> • <code>int32</code> • <code>int64</code> • <code>uint8</code> • <code>uint16</code> • <code>uint32</code> • <code>uint64</code> • <code>logical</code>

CQL Data Type	CQL Data Type Description	Data Type of MATLAB Table Variable for Data Import	Data Type of MATLAB Table Variable for Data Export
tuple<Type1, .., TypeN>	Group of unnamed but typed fields	<p>A 1-by-n MATLAB table, where n is the number of fields in the tuple. The variable names are Var1, Var2, Var3, and so on, until Var(n). The data type of each variable depends on the Cassandra data types defined in the tuple.</p> <p>For example, if a Cassandra database column has the tuple<text, smallint, timestamp> data type, then the partitionRead and executeCQL functions convert this data type into a MATLAB table. The table has the Var1 variable as a string array, Var2 as an int16 array, and Var3 as a datetime array. In this case, the data type of the MATLAB table variable is a table.</p>	<p>MATLAB table.</p> <p>The position of each variable in the table determines which field the variable maps to in the tuple. The first variable in the table maps to the first field in the tuple, the second variable maps to the second field, and so on.</p> <p>For example, if the Cassandra database table contains tuple<int, text>, then the MATLAB table must contain a table where the first variable is numeric and the second variable is a string.</p>
user-defined type (UDT)	Group of named fields	<p>A 1-by-n MATLAB table, where n is the number of fields in the UDT. Variable names match the field names of the UDT. The data type of each variable depends on the Cassandra data types defined in the UDT. The data type of the MATLAB table variable is a table.</p>	<p>MATLAB table.</p> <p>The names of the variables in the table must match the names of the UDT fields. The data type of each variable in the table must be compatible with the Cassandra data type of the corresponding UDT field.</p>
uuid	UUID in standard UUID format	string	char, string, or cell array of character vectors
varchar	UTF-8 encoded string	string	char, string, or cell array of character vectors

CQL Data Type	CQL Data Type Description	Data Type of MATLAB Table Variable for Data Import	Data Type of MATLAB Table Variable for Data Export
varint	Arbitrary-precision integer	string	<ul style="list-style-type: none"> • string • cell array of character vectors • double • single • int8 • int16 • int32 • int64 • uint8 • uint16 • uint32 • uint64 • sym

Note For the CQL data type, if the data type is a collection (for example, `list`, `map`, and so on), then the value contains angle brackets (<>). These brackets surround the data types of the items in the collection. For details about valid Cassandra data types, see [CQL Data Types](#).

See Also

[partitionRead](#) | [upsert](#) | [executecql](#)

More About

- “Fill Values for Missing Data from Database Using Apache Cassandra Database C++ Interface” on page 8-9
- “Import Data from Cassandra Database Table Using CQL” on page 8-14

External Websites

- [Apache Cassandra Documentation](#)
- [CQL Data Types](#)

Fill Values for Missing Data from Database Using Apache Cassandra Database C++ Interface

When you import data from a Cassandra database into MATLAB using the `partitionRead` or `executecql` functions of the Apache Cassandra database C++ interface, these functions convert the fill values for missing data from the Cassandra column to the corresponding MATLAB values. This table identifies the data types in a Cassandra database column (Cassandra Query Language, or CQL, data type) and the corresponding fill value for missing data in the MATLAB table.

CQL Data Type	MATLAB Fill Value
ascii	<missing>
bigint	0
blob	[]
boolean	false
counter	0
date	NaN
decimal	<missing>
double	NaN
float	NaN
inet	<missing>
int	0
list<type>	[]
map<keyType, valueType>	0x2 table (empty table)
set<type>	[]
smallint	0
text	<missing>
time	NaN
timestamp	NaN
timeuuid	<missing>
tinyint	0
tuple<Type1, ..., TypeN>	1xn table, where the missing value in each variable is the MATLAB fill value for the corresponding CQL data type
user-defined type (UDT)	1xn table, where the missing value in each variable is the MATLAB fill value for the corresponding CQL data type
uuid	<missing>
varchar	<missing>

CQL Data Type	MATLAB Fill Value
varint	<missing>

See Also

Objects

connection

Functions

apacheCassandra | partitionRead | executecql

More About

- “Explore and Import Data from Cassandra Database Table” on page 8-11
- “Import Data from Cassandra Database Table Using CQL” on page 8-14

External Websites

- [Apache Cassandra Documentation](#)
- [CQL Data Types](#)

Explore and Import Data from Cassandra Database Table

This example shows how to explore the structure of an Apache™ Cassandra® database and import data from a Cassandra database table into MATLAB® using a Cassandra database connection with the Apache Cassandra database C++ interface. The Cassandra database stores database tables according to the partition key. The partition key affects how the data is filtered in the database.

In this example, the Cassandra database contains the `employees_by_job` database table with employee data and the `job_id` partition key.

Create Cassandra Database Connection

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password);
```

Explore Cassandra Database Structure

View the keyspaces in the Cassandra database using the Cassandra database connection. The returned string array contains the keyspaces. A keyspace contains one or more database tables and defines how the database replicates the data in the tables.

```
conn.Keyspaces
```

```
ans = 6x1 string
    "employeeedata"
    "system"
    "system_auth"
    "system_distributed"
    "system_schema"
    "system_traces"
```

Return the names of the Cassandra database tables in the `employeeedata` keyspace. `t` is a string array that contains the names of the database tables in the `employeeedata` keyspace.

```
keyspace = "employeeedata";
t = tablenames(conn,keyspace)
```

```
t = 3x1 string
    "employees_by_id"
    "employees_by_job"
    "employees_by_name"
```

Return the names of the Cassandra database columns in the `employees_by_job` database table.

```
tablename = "employees_by_job";
cols = columninfo(conn,keyspace,tablename);
```

Display the first few names of the Cassandra database columns in the `employees_by_job` database table.

```
head(cols)
```

```
ans=8x4 table
      Name           DataType      PartitionKey      ClusteringColumn
-----
"job_id"           "text"           true              ""
"hire_date"        "date"           false             "DESC"
"employee_id"      "int"            false             "ASC"
"commission_pct"   "double"         false             ""
"department_id"    "int"            false             ""
"email"            "text"           false             ""
"first_name"       "text"           false             ""
"last_name"        "text"           false             ""
```

`cols` is a table with these variables:

- `Name` — Cassandra database column name
- `DataType` — Cassandra Query Language (CQL) data type of the Cassandra database column
- `PartitionKey` — Partition key indicator
- `ClusteringColumn` — Clustering column indicator

The value in the `PartitionKey` variable indicates whether the database column is a partition key. The column `job_id` (job identifier) is a partition key in this database table.

Import Data from Cassandra Database

Import data from the `employees_by_job` database table into MATLAB. This database has data about shop clerks, so use the partition key value `SH_CLERK`.

```
keyValue = "SH_CLERK";
results = partitionRead(conn, keyspace, tablename, ...
    keyValue);
```

Display the first few rows of the returned employee data.

```
head(results)
```

```
ans=8x13 table
      job_id      hire_date      employee_id      commission_pct      department_id      email
-----
"SH_CLERK"      03-Feb-2008      183              NaN              50              "GGEONI"
"SH_CLERK"      13-Jan-2008      199              NaN              50              "DGRANT"
"SH_CLERK"      19-Dec-2007      191              NaN              50              "RPERKINS"
"SH_CLERK"      21-Jun-2007      182              NaN              50              "MSULLIVA"
"SH_CLERK"      21-Jun-2007      198              NaN              50              "DOCONNEL"
"SH_CLERK"      17-Mar-2007      195              NaN              50              "VJONES"
"SH_CLERK"      07-Feb-2007      187              NaN              50              "ACABRIO"
"SH_CLERK"      11-Jul-2006      190              NaN              50              "TGATES"
```

`results` is a table that contains these variables:

- `job_id` — Job identifier
- `hire_date` — Hire date
- `employee_id` — Employee identifier
- `commission_pct` — Commission percentage
- `department_id` — Department identifier
- `email` — Email address
- `first_name` — First name
- `last_name` — Last name
- `manager_id` — Manager identifier
- `office` — Office location (table that contains two variables for the building and room)
- `performance_ratings` — Performance ratings
- `phone_number` — Phone number
- `salary` — Salary

Close Cassandra Database Connection

```
close(conn)
```

See Also

`apacheCassandra` | `tablenames` | `columninfo` | `partitionRead` | `close`

Related Examples

- “Import Data from Cassandra Database Table Using CQL” on page 8-14
- “Fill Values for Missing Data from Database Using Apache Cassandra Database C++ Interface” on page 8-9

External Websites

- [Apache Cassandra](#)
- [Apache Cassandra Documentation](#)
- [CQL Data Types](#)

Import Data from Cassandra Database Table Using CQL

This example shows how to import data from an Apache™ Cassandra® database table into MATLAB® using the Cassandra Query Language (CQL) and a Cassandra database connection with the Apache Cassandra database C++ interface.

In this example, you use the `executecql` function to execute a CQL query that filters by a clustering column and limits rows in the query results. Alternatively, you can use the `executecql` function to write non-SELECT CQL statements. For easy data import using the partition key values of a Cassandra database table, use the `partitionRead` function instead.

For this example, the Cassandra database contains the `employees_by_job` database table with employee data and the `job_id` partition key. The `hire_date` database column is a clustering column.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password);
```

Write a CQL query that selects all employees who are programmers or shop clerks, and retrieves their job identifiers, hire dates, and email addresses. Filter the query by those employees hired before April 30, 2006, using the `hire_date` clustering column. Limit the returned data to four rows.

```
query = strcat("SELECT job_id,hire_date,email ", ...
    "FROM employeedata.employees_by_job ", ...
    "WHERE job_id IN ('IT_PROG','SH_CLERK') ", ...
    "AND hire_date < '2006-04-30'", ...
    "LIMIT 4;");
```

Execute the CQL query using the Cassandra database connection and display the results.

```
results = executecql(conn,query)
```

```
results=4x3 table
      job_id      hire_date      email
      _____      _____      _____
      "IT_PROG"      05-Feb-2006      "VPATABAL"
      "IT_PROG"      03-Jan-2006      "AHUNOLD"
      "IT_PROG"      25-Jun-2005      "DAUSTIN"
      "SH_CLERK"      24-Apr-2006      "AWALSH"
```

`results` is a table with the `job_id`, `hire_date`, and `email` variables. The `hire_date` variable is a datetime array and the `job_id` and `email` variables are string arrays.

Close the Cassandra database connection.

```
close(conn)
```


See Also

[apacheCassandra](#) | [executecql](#) | [close](#)

Related Examples

- “Explore and Import Data from Cassandra Database Table” on page 8-11
- “Fill Values for Missing Data from Database Using Apache Cassandra Database C++ Interface” on page 8-9

External Websites

- [Apache Cassandra](#)
- [Apache Cassandra Documentation](#)
- [CQL Reference Documentation](#)
- [CQL Data Types](#)

Export MATLAB Data into Cassandra Database

This example shows how to export data from a MATLAB® table into an Apache™ Cassandra® database using a Cassandra database connection with the Apache Cassandra database C++ interface.

In this example, the Cassandra database includes the `employees_by_job` database table, which contains employee data and the `job_id` partition key.

Create Cassandra Database Connection

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password);
```

Explore Data from Cassandra Database

Return the names of the Cassandra database tables in the `employeedata` keyspace. `t` is a string array that contains the names of these tables.

```
keyspace = "employeedata";
t = tablenames(conn,keyspace)
```

```
t = 3x1 string
    "employees_by_id"
    "employees_by_job"
    "employees_by_name"
```

Import employee data into MATLAB from the `employees_by_job` table in the `employeedata` keyspace by using the Cassandra database connection.

```
keyspace = "employeedata";
tablename = "employees_by_job";
results = partitionRead(conn,keyspace,tablename);
```

Display the last few rows of the imported employee data.

```
tail(results)
```

```
ans=8x13 table
    job_id      hire_date      employee_id      commission_pct      department_id      email
    _____  _____  _____  _____  _____  _____
    "SH_CLERK"   27-Jan-2004      184              NaN                50              "NSARCHAN"
    "MK_REP"     17-Aug-2005      202              0.25               20              "PFAY"
    "PU_CLERK"   10-Aug-2007      119              NaN                30              "KCOLMENA"
    "PU_CLERK"   15-Nov-2006      118              NaN                30              "GHIMURO"
    "PU_CLERK"   24-Dec-2005      116              NaN                30              "SBAIDA"
    "PU_CLERK"   24-Jul-2005      117              NaN                30              "STOBIAS"
```

"PU_CLERK"	18-May-2003	115	NaN	30	"AKH00"
"AC_ACCOUNT"	07-Jun-2002	206	NaN	110	"WGIETZ"

results is a table that contains these variables:

- job_id — Job identifier
- hire_date — Hire date
- employee_id — Employee identifier
- commission_pct — Commission percentage
- department_id — Department identifier
- email — Email address
- first_name — First name
- last_name — Last name
- manager_id — Manager identifier
- office — Office location (table that contains two variables for the building and room)
- performance_ratings — Performance ratings
- phone_number — Phone number
- salary — Salary

Display the CQL data types of the columns in the employees_by_job database table.

```
cols = columninfo(conn, keyspace, tablename);
cols(:,1:2)
```

```
ans=13x2 table
```

Name	DataType
"job_id"	"text"
"hire_date"	"date"
"employee_id"	"int"
"commission_pct"	"double"
"department_id"	"int"
"email"	"text"
"first_name"	"text"
"last_name"	"text"
"manager_id"	"int"
"office"	"office"
"performance_ratings"	"list<int>"
"phone_number"	"text"
"salary"	"int"

Insert Data from MATLAB into Cassandra Database

Create a table of data representing one employee to insert into the Cassandra database. Specify the names of the variables. Create a table for the office information. Then, create a table with the employee information that contains the nested table of office information. Set the names of the variables.

```
varnames = ["job_id" "hire_date" "employee_id" ...
            "commission_pct" "department_id" "email" "first_name" ...
```

```

        "last_name" "manager_id" "office" "performance_ratings" ...
        "phone_number" "salary"];
office = table("South",160, ...
    'VariableNames',["building" "room"]);
data = table("IT_ADMIN",datetime('today'),301,0.25,30,"SMITH123", ...
    "Alex","Smith",114,office,{[4 5]},"515.123.2345",3000);
data.Properties.VariableNames = varnames;

```

Insert the employee information into the Cassandra database.

```
upsert(conn, keyspace, tablename, data)
```

Display the inserted data by importing it into MATLAB using the partition key IT_ADMIN. The employees_by_job table contains a new row.

```
keyValue = "IT_ADMIN";
results = partitionRead(conn, keyspace, tablename, keyValue)
```

```
results=1x13 table
    job_id      hire_date      employee_id      commission_pct      department_id      email
    _____      _____      _____      _____      _____      _____
    "IT_ADMIN"      07-Oct-2020      301              0.25              30              "SMITH123"
```

Update Data in Cassandra Database

Update the email variable in the new row of employee information.

```
results.email = "SMITH456";
upsert(conn, keyspace, tablename, results)
```

Display the updated data by importing it into MATLAB. The row contains the updated data in the email variable of the employees_by_job table.

```
results = partitionRead(conn, keyspace, tablename, keyValue)
```

```
results=1x13 table
    job_id      hire_date      employee_id      commission_pct      department_id      email
    _____      _____      _____      _____      _____      _____
    "IT_ADMIN"      07-Oct-2020      301              0.25              30              "SMITH456"
```

Close Cassandra Database Connection

```
close(conn)
```

See Also

apacheCassandra | tablename | upsert | partitionRead | close

Related Examples

- “Explore and Import Data from Cassandra Database Table” on page 8-11
- “Import Data from Cassandra Database Table Using CQL” on page 8-14

- “Convert CQL Data Types to MATLAB Data Types Using Apache Cassandra Database C++ Interface” on page 8-2

External Websites

- [Apache Cassandra](#)
- [Apache Cassandra Documentation](#)

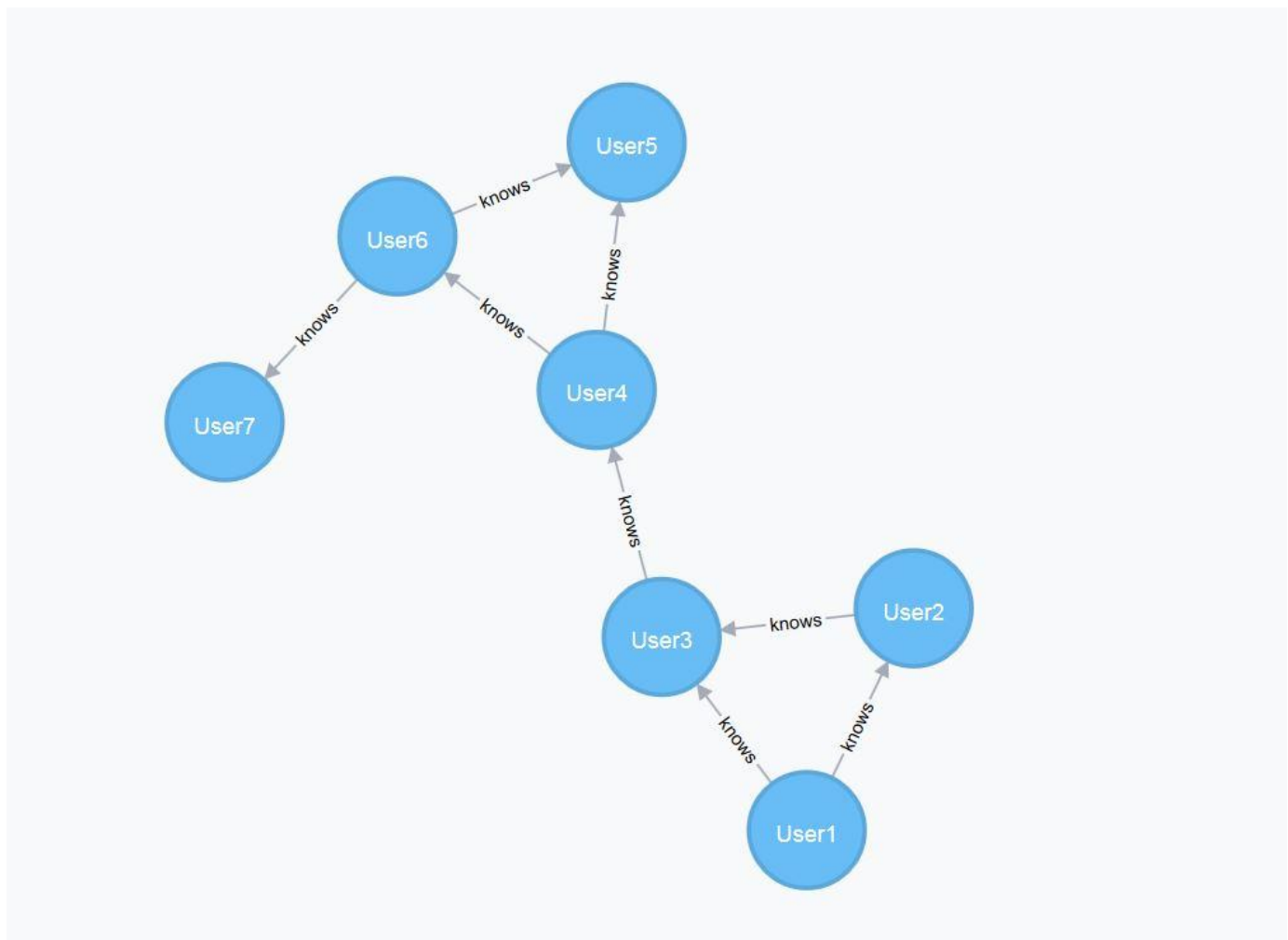
Neo4j Topics

Explore Graph Database Structure

This example shows how to traverse a graph and explore its structure using the MATLAB® interface to Neo4j®. For details about the MATLAB interface to Neo4j, see “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type knows.

The local machine hosts the Neo4j database with the port number 7474, user name `neo4j`, and password `matlab`. This figure provides a visual representation of the data in the database.



Connect to Neo4j Database

Create a Neo4j connection object `neo4jconn` using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';  
username = 'neo4j';
```



```
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the Message property of the Neo4j connection object neo4jconn. The blank Message property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Explore Structure of Entire Graph

Find all the node labels in the Neo4j database using the Neo4j connection object neo4jconn.

```
nlabels = nodeLabels(neo4jconn)
```

```
nlabels = 1x1 cell array
    {'Person'}
```

Find all the relationship types in the Neo4j database.

```
reltypes = relationTypes(neo4jconn)
```

```
reltypes = 1x1 cell array
    {'knows'}
```

Find the property keys in the Neo4j database.

```
propkeys = propertyKeys(neo4jconn)
```

```
propkeys = 15x1 cell array
    {'Name' }
    {'property' }
    {'title' }
    {'Description' }
    {'EndNodes' }
    {'Location' }
    {'EndDate' }
    {'Address' }
    {'Project' }
    {'Department' }
    {'StartDate' }
    {'Title' }
    {'Date' }
    {'Weight' }
    {'name' }
```

Search for Nodes

Search for all the nodes with the node label Person. The nodesinfo output argument contains node labels, node data, and the Neo4jNode objects for each matched node.

```
nlabel = 'Person';
nodesinfo = searchNode(neo4jconn,nlabel)
nodesinfo=7x3 table
      NodeLabels      NodeData      NodeObject
-----
0      'Person'      [1x1 struct]  [1x1 database.neo4j.http.Neo4jNode]
1      'Person'      [1x1 struct]  [1x1 database.neo4j.http.Neo4jNode]
2      'Person'      [1x1 struct]  [1x1 database.neo4j.http.Neo4jNode]
3      'Person'      [1x1 struct]  [1x1 database.neo4j.http.Neo4jNode]
4      'Person'      [1x1 struct]  [1x1 database.neo4j.http.Neo4jNode]
5      'Person'      [1x1 struct]  [1x1 database.neo4j.http.Neo4jNode]
9      'Person'      [1x1 struct]  [1x1 database.neo4j.http.Neo4jNode]
```

Search for the node with the node identifier 2. The `nodeinfo` output argument contains the node identifier, node data, and node labels for the node with node identifier 2.

```
nodeid = 2;
nodeinfo = searchNodeByID(neo4jconn,nodeid)
nodeinfo =
  Neo4jNode with properties:
      NodeID: 2
      NodeData: [1x1 struct]
      NodeLabels: 'Person'
```

Search for Relationships

Search for incoming relationship types that belong to the node `nodeinfo`.

```
nodereltypes = nodeRelationTypes(nodeinfo,'in')
nodereltypes = 1x1 cell array
    {'knows'}
```

Search for the degree of all incoming relationships that belong to the node `nodeinfo`.

```
degree = nodeDegree(nodeinfo,'in')
degree = struct with fields:
    knows: 1
```

Search for the relationship with the node identifier 4.

```
relationid = 4;
relationinfo = searchRelationByID(neo4jconn,relationid)
relationinfo =
  Neo4jRelation with properties:
```

```

RelationID: 4
RelationData: [1x1 struct]
StartNodeID: 3
RelationType: 'knows'
EndNodeID: 5

```

Search for all incoming relationships that belong to the node `nodeinfo`. The `relinfo` output argument contains data about the start and end nodes and all matched relationships from the origin node.

```
relinfo = searchRelation(neo4jconn,nodeinfo,'in')
```

```

relinfo = struct with fields:
    Origin: 2
    Nodes: [2x3 table]
    Relations: [1x5 table]

```

Retrieve Entire Graph

Retrieve the entire graph using node labels `nlabels`.

```
graphinfo = searchGraph(neo4jconn,nlabels)
```

```

graphinfo = struct with fields:
    Nodes: [7x3 table]
    Relations: [8x5 table]

```

`graphinfo` contains node data for all start and end nodes for each matched relationship. `graphinfo` also contains relationship data for each matched relationship.

Close Database Connection

```
close(neo4jconn)
```

See Also

[neo4j](#) | [nodeLabels](#) | [relationTypes](#) | [propertyKeys](#) | [searchNodeByID](#) | [searchNode](#) | [nodeRelationTypes](#) | [nodeDegree](#) | [searchRelation](#) | [searchRelationByID](#)

More About

- “Search Graph Database” on page 9-9
- “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

External Websites

- [Neo4j Documentation](#)

Graph Database Workflow for Neo4j Database Interfaces

You can interact with graph data stored in a Neo4j database using the MATLAB interface to Neo4j or the Database Toolbox Interface for Neo4j Bolt Protocol. The MATLAB interface to Neo4j uses the REST API to create a database connection. The Database Toolbox Interface for Neo4j Bolt Protocol uses the Bolt protocol to make the connection.

The MATLAB interface to Neo4j enables you to:

- Create a Neo4j database connection using the REST API.
- Import graph data from a Neo4j database into MATLAB.
- Perform graph network analysis by creating a directed graph from the imported graph data.
- Create, update, and delete nodes and relationships in the graph database.
- Update node labels and properties and update relationship properties.
- Export a directed graph from MATLAB into a Neo4j database.
- Execute Cypher[®] queries on the Neo4j database, if you are familiar with the Cypher query language.
- Close the database connection.

The Database Toolbox Interface for Neo4j Bolt Protocol enables you to use the same functionality and the same workflow to interact with graph data stored in a Neo4j database.

About Neo4j Graph Databases

A graph database stores data using a graph data model. This model consists of nodes and relationships. A relationship describes how two or more nodes are related to each other.

Nodes can have one or more node labels and property keys, or zero labels and property keys. Neo4j assigns unique identifiers to nodes and relationships.

Relationships are always directed and have a relationship type. A relationship always has a start node and an end node. A node can have incoming and outgoing relationships. Two nodes can have multiple relationships between them.

For details about graphs, see “Directed and Undirected Graphs”. For details about the Neo4j database, see Why Graph Databases?

Neo4j Graph Database Workflow

This workflow shows how to connect to a Neo4j database, search and update the graph database, store a directed graph, and perform graph network analysis.

- 1 Connect to a Neo4j database using `neo4j`.
- 2 Search the graph database.

Conduct a general search in the graph database using any of these functions:

- `nodeLabels`
- `propertyKeys`

- `relationTypes`
- `searchGraph`

Or, conduct a targeted search in the graph database using any of these functions:

- `searchNode`
- `searchNodeByID`
- `searchRelation`
- `searchRelationByID`
- `nodeDegree`
- `nodeRelationTypes`

3 Update the graph database.

Create nodes and relationships using these functions:

- `createNode`
- `createRelation`

Update nodes and relationships using these functions:

- `addNodeLabel`
- `removeNodeLabel`
- `removeNodeProperty`
- `removeRelationProperty`
- `setNodeProperty`
- `setRelationProperty`
- `updateNode`
- `updateRelation`

Delete nodes and relationships using these functions:

- `deleteNode`
- `deleteRelation`

4 Export a directed graph from MATLAB into a Neo4j database by using the `storeDigraph` function.

5 To perform graph network analysis, you can convert output structures to `digraph` objects using `neo4jStruct2Digraph`. For details, see “Directed and Undirected Graphs”.

Or, if you know the Cypher query language, you can execute a Cypher query using `executeCypher`. For details, see Cypher Query Language.

6 Close the database connection using the `close` function.

Advantage of Database Toolbox Interface for Neo4j Bolt Protocol

You can connect to a Neo4j database using the REST API or the Bolt protocol. The Bolt protocol provides the advantage of sending binary data instead of a JSON payload using the REST API. Binary data is smaller than a JSON payload. Sending data of a smaller size generally improves performance when you use the Bolt protocol.

To use the Bolt protocol, you must install the Database Toolbox Interface for Neo4j Bolt Protocol. For details, see “Database Toolbox Interface for Neo4j Bolt Protocol Installation” on page 9-37. For details about connecting to a Neo4j database using either interface, see the `neo4j` function.

See Also

`neo4j` | `neo4jStruct2Digraph` | `digraph`

More About

- “Find Friends of Friends in Social Neighborhood” on page 9-32
- “Visualize Breadth-First and Depth-First Search”
- “Directed and Undirected Graphs”
- “Update Friend Information in Social Neighborhood” on page 9-11
- “Add and Query Group of Colleagues in Social Neighborhood” on page 9-14
- “Search Graph Database” on page 9-9
- “Error Messages for Neo4j Database Interfaces” on page 9-20

External Websites

- [Neo4j Documentation](#)
- [Cypher Query Language](#)

Search Graph Database

Search a Neo4j graph database using functions provided by the MATLAB interface to Neo4j and the Database Toolbox Interface for Neo4j Bolt Protocol. You can explore the graph data and perform graph network analysis using MATLAB directed graphs.

Search Functionality

Search graph data in a Neo4j graph database using different parts of the graph:

- Search for one or more nodes using `searchNode`. Search for a node with a specific identifier using `searchNodeByID`.
- Search for relationships from an origin node using `searchRelation`.
- Search for the entire graph database or a subgraph using `searchGraph`.

To access the part of the graph database that you want to analyze, combine these functions and explore the graph data in the output arguments.

General and Targeted Search Workflows

You can search a Neo4j graph database in a general or targeted way. A general search starts from a subgraph or the entire graph. A targeted search starts from an origin node and traverses its relationships.

After finding a part of the graph, you can create a MATLAB directed graph and perform graph network analysis.

Conduct General Search

- 1 Conduct a general search for a subgraph using `searchGraph`.

For example, to find the subgraph `graphinfo`, enter this code, which assumes a successful Neo4j database connection `neo4jconn`. The `graphinfo` output argument is a directed graph.

```
nlabel = {'Person'};

graphinfo = searchGraph(neo4jconn,nlabel, ...
    'DataReturnFormat','digraph');
```

- 2 Perform graph network analysis using the `digraph` object `G`. For details, see “Directed and Undirected Graphs”.

For example, determine the shortest path between nodes using `distances`.

```
d = distances(G);
```

Or, explore the graph data by executing the `searchGraph` function without the `'DataReturnFormat'` name-value pair argument and accessing the output structure `graphinfo`.

Conduct Targeted Search

- 1 To start your search, find the origin node using `searchNode` or `searchNodeByID`.

For example, to find the origin node `nodeinfo`, enter this code, which assumes a successful Neo4j database connection `neo4jconn` and the node identifier 2.

```
nodeinfo = searchNodeByID(neo4jconn,2);
```

- 2 Search for graph data by using the origin node and `searchRelation`. Or, if you know the relationship identifier, then use the `searchRelationByID` function.

For example, this code assumes that you are searching for incoming relationships. The `reinfo` output argument is a directed graph.

```
reinfo = searchRelation(neo4jconn,nodeinfo,'in','DataReturnFormat','digraph');
```

- 3 Perform graph network analysis using the `digraph` object `G`. For details, see “Directed and Undirected Graphs”.

For example, determine the shortest path between nodes using `distances`.

```
d = distances(G);
```

Or, explore node information by accessing the output structure `nodeinfo`. Also, explore relationship information by executing the `searchRelation` function without the 'DataReturnFormat' name-value pair argument and accessing the output structure `reinfo`.

See Also

`searchNode` | `searchNodeByID` | `searchRelation` | `searchGraph` | `nodeDegree`

More About

- “Explore Graph Database Structure” on page 9-2
- “Find Shortest Path Between People in Social Neighborhood” on page 9-27
- “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6
- “Error Messages for Neo4j Database Interfaces” on page 9-20

External Websites

- Neo4j Documentation

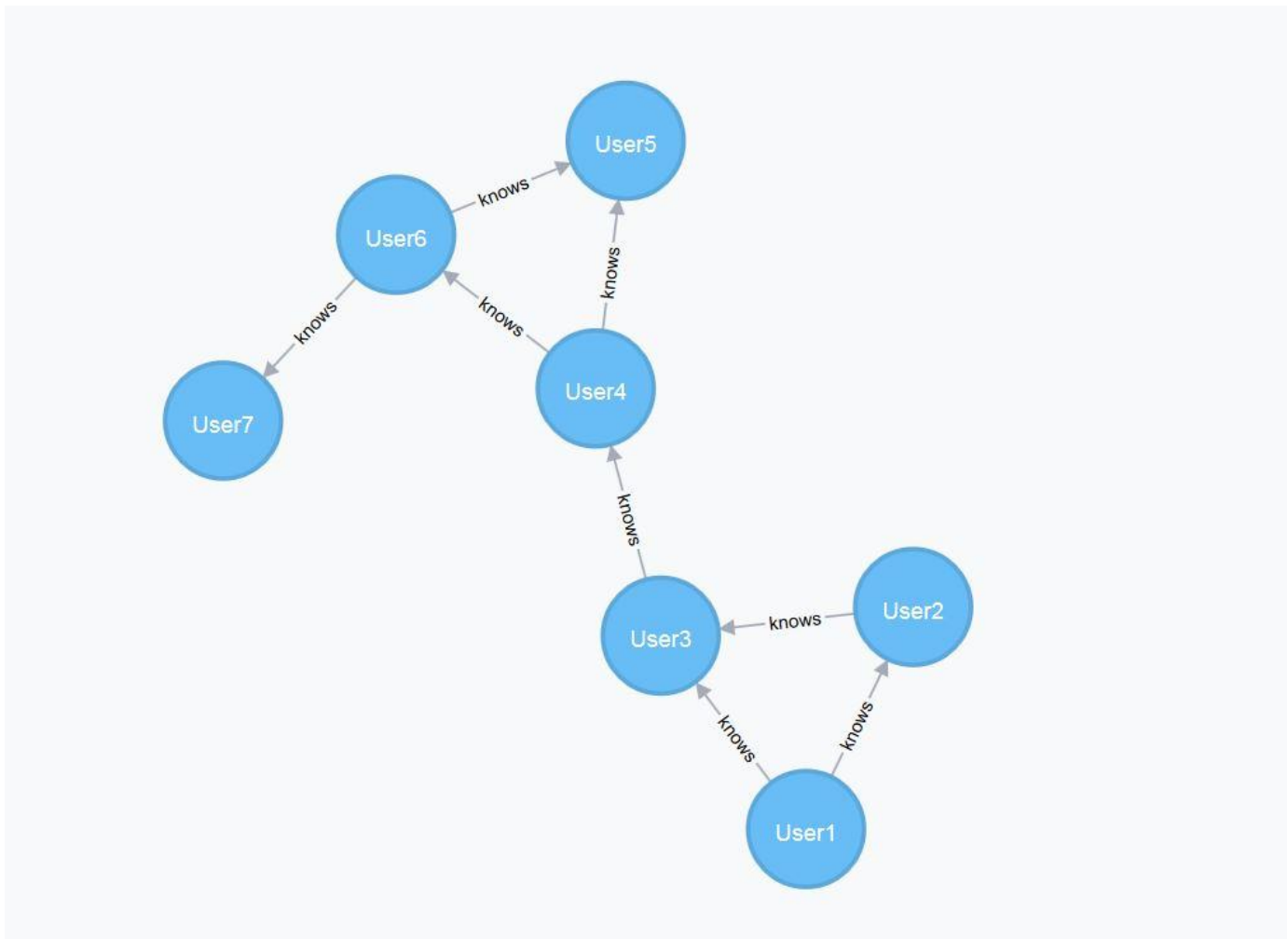
Update Friend Information in Social Neighborhood

This example shows how to create, update, and delete information in a social neighborhood, which is represented by a Neo4j® database, using the MATLAB® interface to Neo4j.

For details about the MATLAB interface to Neo4j, see “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6.

Assume that you have graph data stored in a Neo4j database that represents the social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type `knows`.

The local machine hosts the Neo4j database with the port number 7474, user name `neo4j`, and password `matlab`. This figure provides a visual representation of the data in the database.



Connect to Neo4j Database

Create a Neo4j connection object `neo4jconn` using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Add Two Friends to Social Neighborhood

Create two nodes in the database using the Neo4j database connection. Use the `'Labels'` name-value pair argument to specify the label `Person` for each node.

```
label = 'Person';
user8 = createNode(neo4jconn,'Labels',label);
user9 = createNode(neo4jconn,'Labels',label);
```

Search for the node with the label `Person` and the property key name set to the value `User7` by using the Neo4j database connection.

```
nlabel = 'Person';
user7 = searchNode(neo4jconn,nlabel,'PropertyKey','name', ...
    'PropertyValue','User7');
```

Create two relationships using the Neo4j database connection. Specify the relationship types as `works with` and `studies with`. The two relationships are:

- `User8` works with `User7`
- `User8` studies with `User9`

`relationinfo` is a table that contains the relationship and node information.

```
startnode = [user8,user8];
endnode = [user7,user9];
relationtype = {'works with','studies with'};
relationinfo = createRelation(neo4jconn,startnode,endnode,relationtype);
```

Update Node Information for Added Friend

Update the properties of the node `User8`. Create a table with one row that contains the name and job title for this person. `nodeinfo` is a `Neo4jNode` object.

```
properties = table("User8","Analyst","VariableNames',{'Name','Title'});
nodeinfo = setNodeProperty(neo4jconn,user8,properties);
```

Add the node label `Student` to `User9`.

```
labels = 'Student';
nodeinfo = addNodeLabel(neo4jconn,user9,labels);
```

Update Relationship Information for Added Friends

Create a table that defines the relationship properties. Here, User8 works with User7 in the workplace, and User8 studies with User9 in the library. Also, User8 started working with User7 on January 2, 2017, and User8 started studying with User9 on March 6, 2017.

```
properties = table(["Workplace";"Library"],["01/02/2017";"03/06/2017"], ...
  'VariableNames',{ 'Location', 'Date' });
```

Update both relationships with these properties. `relationinfo` is a table that contains the updated relationships.

```
relations = relationinfo.RelationObject;
relationinfo = setRelationProperty(neo4jconn,relations,properties);
```

Delete Relationship for Added Friend

Delete the relationship that connects User8 to User7.

```
relation = relations(1);
deleteRelation(neo4jconn,relation)
```

Delete Friends

Delete the added nodes and any associated relationships.

```
nodes = [user8,user9];
deleteNode(neo4jconn,nodes, 'DeleteRelations',true)
```

Close Database Connection

```
close(neo4jconn)
```

See Also

[Neo4jNode](#) | [deleteNode](#) | [neo4j](#) | [deleteRelation](#) | [createNode](#) | [createRelation](#) | [addNodeLabel](#) | [setNodeProperty](#) | [setRelationProperty](#) | [close](#)

More About

- “Find Shortest Path Between People in Social Neighborhood” on page 9-27
- “Find Friends of Friends in Social Neighborhood” on page 9-32
- “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

External Websites

- [Neo4j Documentation](#)

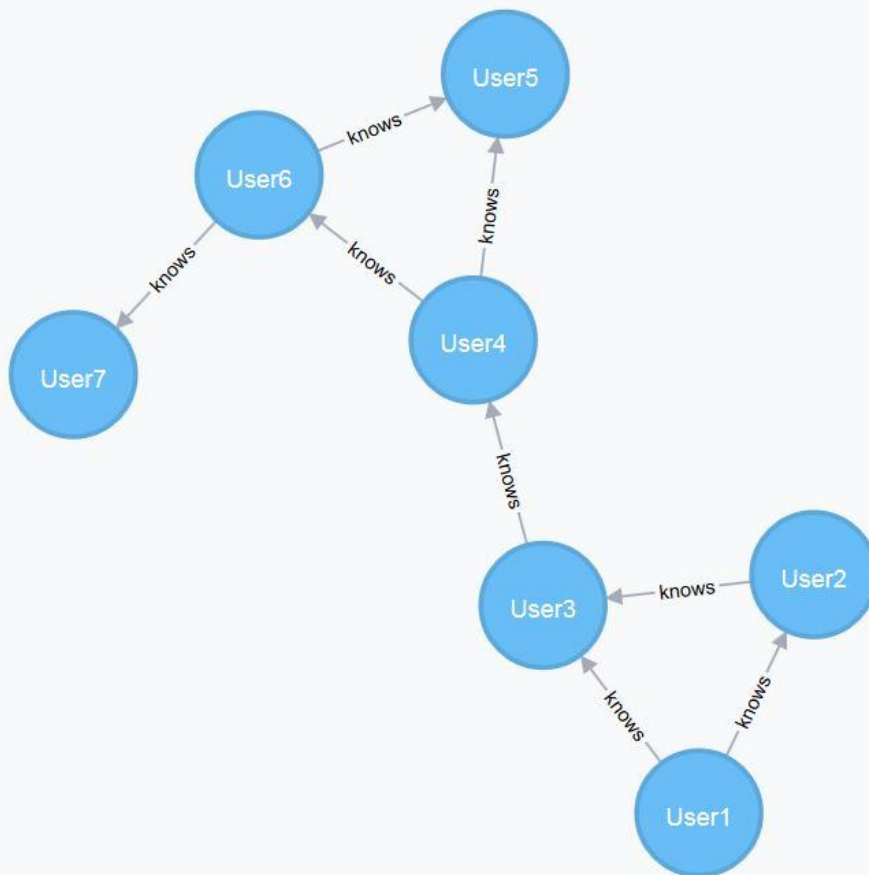
Add and Query Group of Colleagues in Social Neighborhood

This example shows how to add a group of colleagues, stored as a directed graph, to a group of friends in a social neighborhood, stored as nodes and relationships in a Neo4j® database. The example then shows how to query the graph in the database by using the Cypher® query language, which enables you to create custom queries.

For details about the MATLAB® interface to Neo4j, see “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6.

Assume that you have graph data stored in a Neo4j database that represents the social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type `knows`.

The local machine hosts the Neo4j database with the port number 7474, user name `neo4j`, and password `matlab`. This figure provides a visual representation of the data in the database.



Connect to Neo4j Database

Create a Neo4j connection object `neo4jconn` using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';  
username = 'neo4j';  
password = 'matlab';  
  
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Create Directed Graph

Define a group of four colleagues by creating a directed graph in MATLAB. Create a `digraph` object that has four nodes and three edges.

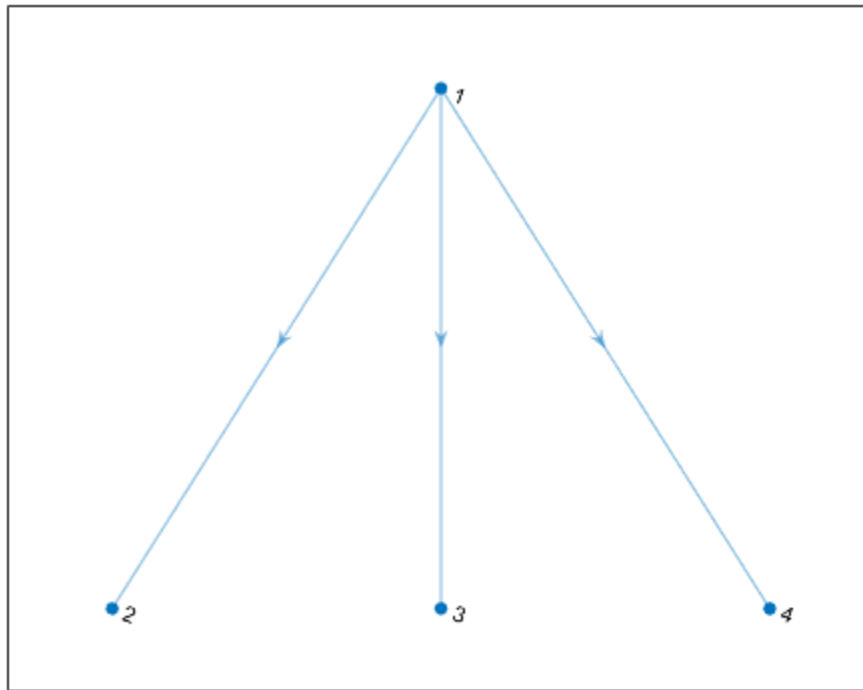
```
s = [1 1 1];  
t = [2 3 4];  
G = digraph(s,t);
```

Specify names for the nodes.

```
G.Nodes.name = {'User8'; 'User9'; 'User10'; 'User11'};
```

Plot the digraph to view the nodes and edges.

```
plot(G)
```



Store Directed Graph in Neo4j Database

Store the directed graph as a Neo4j graph. Specify two labels for all nodes in the resulting Neo4j graph by using the `GlobalNodeLabel` name-value pair argument. Also, specify the type `works with` for all relationships in the resulting Neo4j graph by using the `GlobalRelationType` name-value pair argument.

```
graphinfo = storeDigraph(neo4jconn,G, ...
    'GlobalNodeLabel',{'Colleague','Person'}, ...
    'GlobalRelationType','works with');
```

Display the node labels of the first node in the graph.

```
graphinfo.Nodes.NodeLabels{1}
```

```
ans = 2x1 cell array
    {'Person' }
    {'Colleague'}
```

The result is a cell array of character vectors. Each character vector is a node label for the first node.

Display the relationships in the graph.

```
graphinfo.Relations
```

```
ans=3x5 table
    StartNodeID  RelationType  EndNodeID  RelationData  RelationObject
```

20	31	'works with'	8	[1x1 struct]	[1x1 database.neo4j.http.N
23	31	'works with'	32	[1x1 struct]	[1x1 database.neo4j.http.N
24	31	'works with'	33	[1x1 struct]	[1x1 database.neo4j.http.N

Relations is a table that contains these variables:

- Start node identifier
- Relationship type
- End node identifier
- Relationship data
- Neo4jRelation object

Connect Group of Colleagues to Friends in Existing Graph

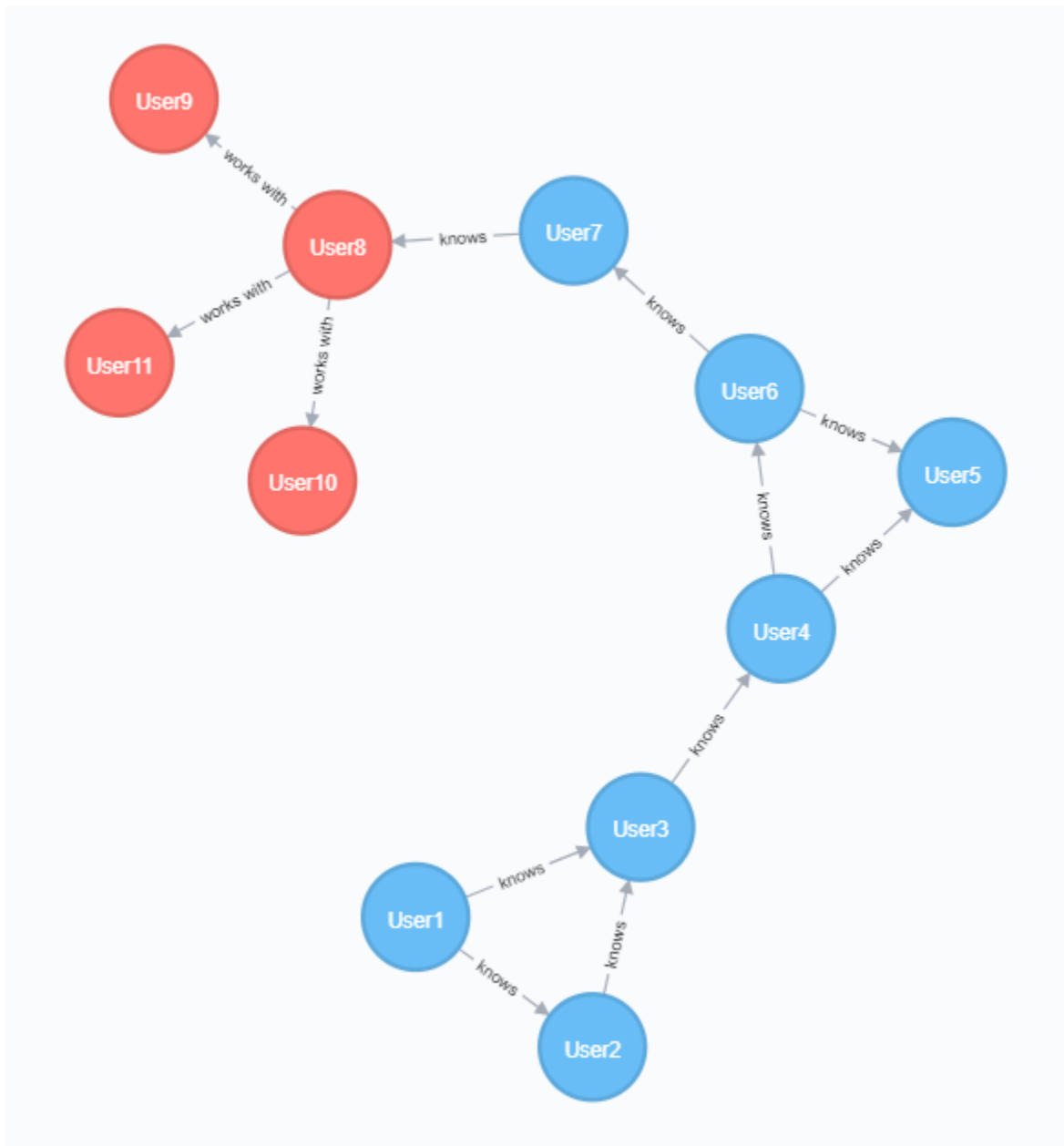
Search for the nodes with the node label Person and the property key name set to the values User7 and User8 by using the Neo4j database connection.

```
nlabel = 'Person';
user7 = searchNode(neo4jconn,nlabel,'PropertyKey','name', ...
  'PropertyValue','User7');
user8 = searchNode(neo4jconn,nlabel,'PropertyKey','name', ...
  'PropertyValue','User8');
```

Add a relationship between the nodes User7 and User8 to connect the group of colleagues to the group of friends.

```
relationtype = 'knows';
relation = createRelation(neo4jconn,user7,user8,relationtype);
```

Display the resulting graph in the Neo4j database.



Execute Cypher Query on Neo4j Database

Create a Cypher query to find the people who work with people User7 knows. Display the names of those people.

```

query = ['MATCH (:Person {name: "User7"})-[:knows]->(:Person)-[:`works with`]' ...
        '->(potentialContact:Person) RETURN potentialContact.name'];
results = executeCypher(neo4jconn,query)

```

```

results=3x1 table
  potentialContact_name
  _____

```



```
'User11'  
'User10'  
'User9'
```

User9, User10, and User11 all work with someone that User7 knows. User7 knows User8, who works with User9, User10, and User11.

Close Database Connection

```
close(neo4jconn)
```

See Also

`neo4j | storeDigraph`

More About

- “Search Graph Database” on page 9-9
- “Explore Graph Database Structure” on page 9-2

External Websites

- [Neo4j Documentation](#)
- [Cypher Query Language](#)

Error Messages for Neo4j Database Interfaces

The MATLAB interface to Neo4j, the Database Toolbox Interface for Neo4j Bolt Protocol, and the Neo4j database return error messages.

The Neo4j database error messages always have a status code that starts with `Neo.ClientError`. To troubleshoot these errors, consult the Neo4j Documentation.

The MATLAB interface to Neo4j and the Database Toolbox Interface for Neo4j Bolt Protocol return error messages in plain text. This table describes how to address common errors you can encounter while working with both interfaces.

Error Message	Probable Cause	Resolution
Invalid connection.	The Neo4j database connection is invalid.	Connect to the Neo4j database using <code>neo4j</code> .
Unable to connect. Please try again.	The Neo4j database connection is invalid.	Connect to the Neo4j database using <code>neo4j</code> .
No Nodes found with matching criteria.	The search cannot find nodes for the specified node label or property keys and values.	Verify the node label or property keys and values. Then, run <code>searchNode</code> .
Unable to find "relationship" relationships for node with id "node identifier" in database.	The search cannot find relationships for the specified relationships and node in the Neo4j database.	Verify the origin node and direction. Then, run <code>searchRelation</code> .
No node labels found.	The Neo4j database has no node labels.	Open the Neo4j database and add node labels. For details, see the Neo4j Operations Manual in the Neo4j Documentation. Then, run <code>nodeLabels</code> .
No relationship types found.	The Neo4j database has no relationship types.	Open the Neo4j database and add relationship types. For details, see the Neo4j Operations Manual in the Neo4j Documentation. Then, run <code>relationshipTypes</code> .
No property keys found.	The Neo4j database has no property keys.	Open the Neo4j database and add property keys. For details, see the Neo4j Operations Manual in the Neo4j Documentation. Then, run <code>propertyKeys</code> .
Unable to execute Cypher query.	The Cypher query is invalid.	Verify the Cypher query. Then, run <code>executeCypher</code> . For details about writing Cypher queries, see Cypher Query Language.

Error Message	Probable Cause	Resolution
Unable to find one or more of the specified nodes in the database.	When you update or delete nodes, the specified node does not exist in the Neo4j database. Or, when you create a relationship, the specified start node or end node does not exist in the Neo4j database.	Find nodes in the Neo4j database using the <code>searchNode</code> function.
Unable to find one or more of the specified relations in the database.	When you update or delete relationships, the specified relationship does not exist in the Neo4j database.	Find relationships in the Neo4j database using the <code>searchRelation</code> function.
One or more nodes have relations. Explicitly delete relations. Or, to delete both nodes and all associated relations, set 'DeleteRelations' to true.	When you delete nodes, the specified nodes have associated relationships.	Delete the node and its associated relationships by using this syntax of the <code>deleteNode</code> function: <code>deleteNode(neo4jconn, node, 'DeleteRelations', true)</code>
Duplicate nodes not supported.	When you create or update a node, you cannot specify duplicate nodes.	Remove duplicate nodes from the input arguments of the function you execute.
Duplicate relations not supported.	When you create or update a relationship, you cannot specify duplicate relationships.	Remove duplicate relationships from the input arguments of the function you execute.
Database Toolbox Interface for Neo4j Bolt Protocol has not been installed. Open Add-On Explorer to install the add-on.	You specify the Bolt database connection URL that starts with the <code>bolt://</code> protocol identifier and the Database Toolbox Interface for Neo4j Bolt Protocol is not installed.	Install the Database Toolbox Interface for Neo4j Bolt Protocol. For details, see “Database Toolbox Interface for Neo4j Bolt Protocol Installation” on page 9-37.

See Also

neo4j | searchGraph

More About

- “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6
- “Search Graph Database” on page 9-9
- “Find Friends of Friends in Social Neighborhood” on page 9-32
- “Explore Graph Database Structure” on page 9-2

External Websites

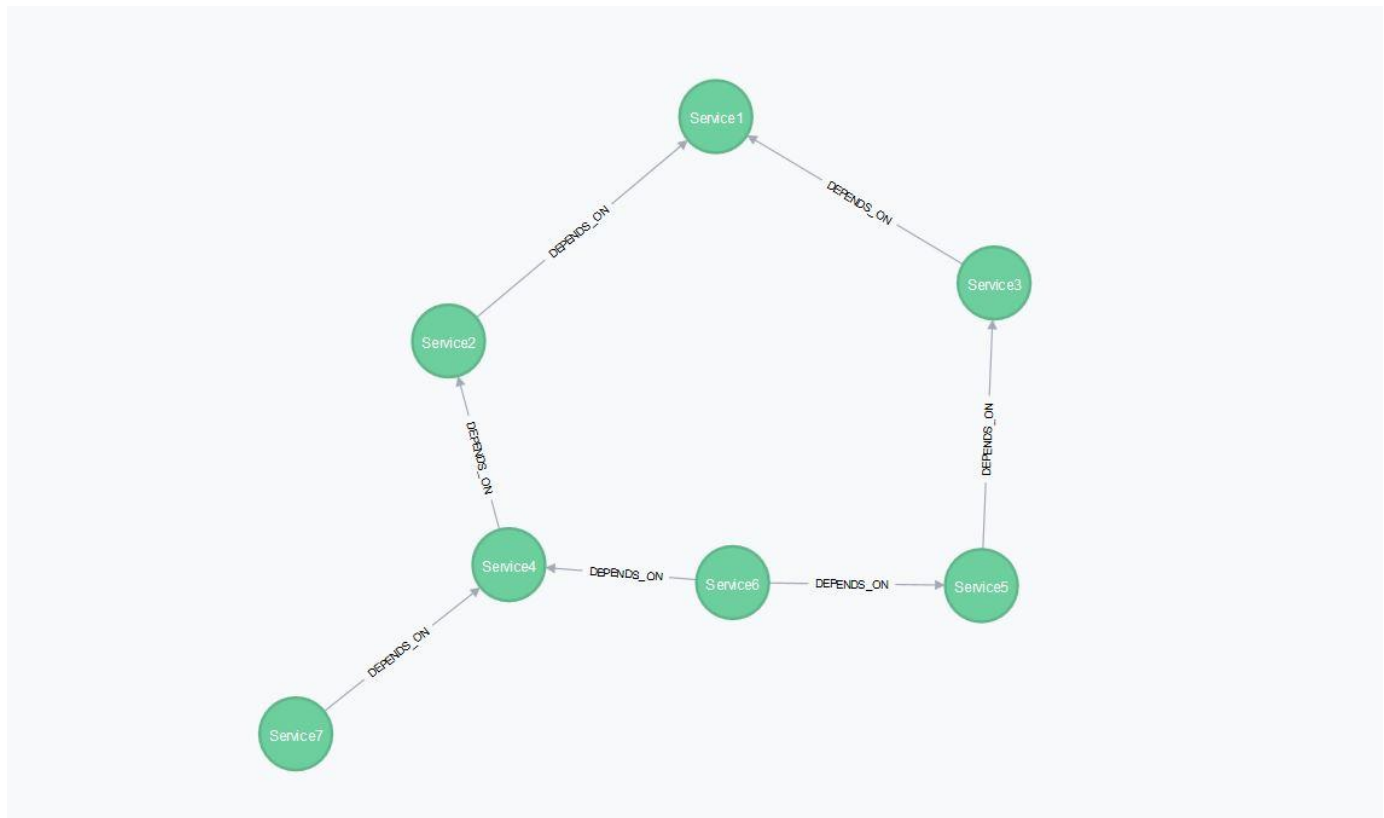
- Neo4j Documentation
- Cypher Query Language

Determine Dependencies of Services in Network

This example shows how to analyze dependencies among services in a network using the MATLAB® interface to Neo4j®. Assume that you have graph data that is stored on a Neo4j database which represents a network. This database has seven nodes and seven relationships. Each node has only one unique property key name with values Service1 through Service7. Each relationship has type `DEPENDS_ON`.

To find the number of services each service depends on, use the MATLAB interface to Neo4j and the `digraph` object. For details about the MATLAB interface to Neo4j, see “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6.

The local machine hosts the Neo4j database with port number 7474, user name `neo4j`, and password `matlab`. For a visual representation of the data in the database, see this figure.



Connect to Neo4j Database

Create a Neo4j connection object `neo4jconn` using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```

url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
  
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Retrieve Entire Graph

Retrieves all relationships of type `DEPENDS_ON` and all nodes associated with each relationship.

```
network_graphdata = searchGraph(neo4jconn,{'DEPENDS_ON'})
```

```
network_graphdata = struct with fields:
```

```
  Nodes: [7×3 table]
```

```
  Relations: [7×5 table]
```

Convert Graph Data to Directed Graph

Using the table `network_graphdata.Nodes`, access the `name` property for each node that appears in the `NodeData` variable of the table.

Assign the table `network_graphdata.Nodes` to `nodestable`.

```
nodestable = network_graphdata.Nodes
```

```
nodestable=7×3 table
```

	NodeLabels	NodeData	NodeObject
6	'Service'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
0	'Service'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
4	'Service'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
2	'Service'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
3	'Service'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
5	'Service'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
1	'Service'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]

Assign the row names for each row to `rownames`.

```
rownames = nodestable.Properties.RowNames
```

```
rownames = 7×1 cell array
```

```
 {'6'}  
 {'0'}  
 {'4'}  
 {'2'}  
 {'3'}  
 {'5'}  
 {'1'}
```

Access the `NodeData` variable from `nodestable` for each row. `nodedata` contains an array of structures.

```
nodedata = [nodetable.NodeData{rownames}]
```

```
nodedata = 1x7 struct array with fields:  
    name
```

To retrieve the name field from each structure, index into the array. `nodenames` is a cell array of character vectors that contains node names.

```
nodenames = {nodedata(:).name}
```

```
nodenames = 1x7 cell array  
    {'Service7'}    {'Service6'}    {'Service1'}    {'Service3'}    {'Service2'}    {'Service4'}
```

Create the `digraph` object `network_graph` using the `neo4jStruct2Digraph` function and the graph `network_graphdata`.

```
network_graph = neo4jStruct2Digraph(network_graphdata, 'NodeNames', nodenames)
```

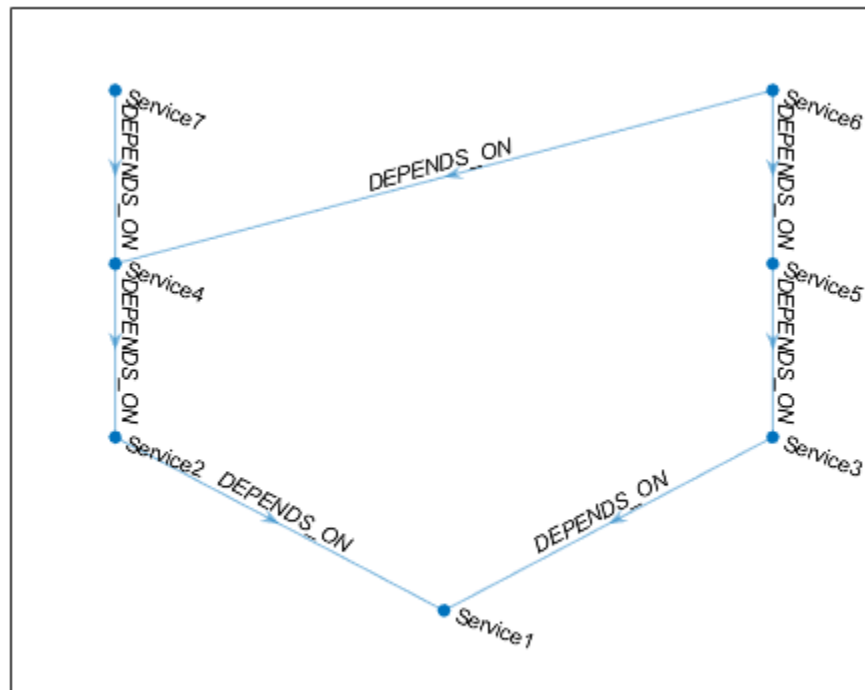
```
network_graph =  
    digraph with properties:
```

```
    Edges: [7x3 table]
```

```
    Nodes: [7x3 table]
```

To see a visual representation of the graph, create a figure that displays the graph `network_graph`.

```
plot(network_graph, 'EdgeLabel', network_graph.Edges.RelationType)
```



Find Dependency Count of Each Service

Find the number of services that each service depends on in the graph `network_graph`. Determine the dependency count for each node by iterating through the nodes in the graph using a for loop. To determine the dependency count, use the `nearest` function.

Create the table `dependency_count` that stores the count of the dependent services for each service. Sort the rows in the table by dependency count in descending order.

```

dependency_count = table;

for i = 1:height(network_graph.Nodes)
    nodeid = network_graph.Nodes.Name(i);
    nearest_node = nearest(network_graph,nodeid,Inf,'Direction','outgoing');
    nearest_length = length(nearest_node);
    dependency_count = [dependency_count; ...
        table(nodeid,nearest_length, ...
            'VariableNames',{'Node','Dependency_count'})];
end

dependency_count = sortrows(dependency_count,-2)
  
```

```

dependency_count=7x2 table
    Node      Dependency_count
    _____
    'Service6'      5
  
```

```
'Service7'      3
'Service4'      2
'Service5'      2
'Service3'      1
'Service2'      1
'Service1'      0
```

Find All Dependencies for Specific Service

Find all the services that the service `Service6` depends on in the graph `network_graph` using the `nearest` function.

```
disp('Service6 depends on the following services:');
Service6 depends on the following services:
nearest(network_graph, 'Service6', Inf, 'Direction', 'outgoing')
ans = 5x1 cell array
    {'Service4'}
    {'Service5'}
    {'Service3'}
    {'Service2'}
    {'Service1'}
```

Close Database Connection

```
close(neo4jconn)
```

See Also

`neo4j` | `searchNode` | `searchRelation` | `nearest`

Related Examples

- “Find Friends of Friends in Social Neighborhood” on page 9-32

More About

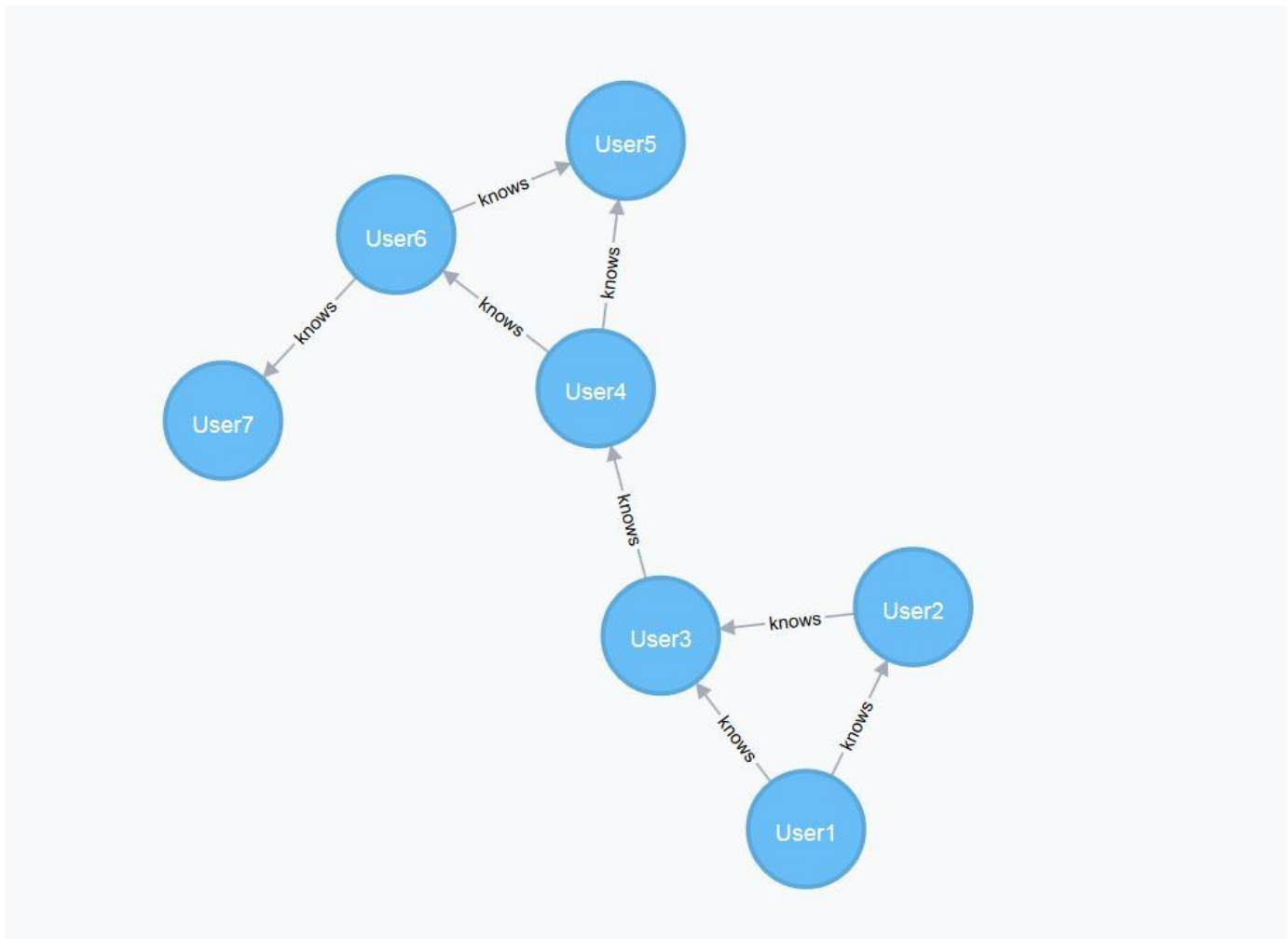
- “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6
- “Directed and Undirected Graphs”

Find Shortest Path Between People in Social Neighborhood

This example shows how to search a social neighborhood to find the shortest path between people, using the MATLAB® interface to Neo4j®. Assume that you have graph data that is stored on a Neo4j database which represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has type `knows`.

To find the shortest path between User1 and User7, use the MATLAB interface to Neo4j and the `digraph` object. For details about the MATLAB interface to Neo4j, see “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6.

The local machine hosts the Neo4j database with port number 7474, user name `neo4j`, and password `matlab`. For a visual representation of the data in the database, see this figure.



Connect to Neo4j Database

Create a Neo4j connection object `neo4jconn` using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the Message property of the Neo4j connection object neo4j conn. The blank Message property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Search Entire Graph

Find all the Person nodes and all the relationships associated with each Person node using searchGraph.

```
social_graphdata = searchGraph(neo4jconn,{'Person'})

social_graphdata = struct with fields:
    Nodes: [7×3 table]
    Relations: [8×5 table]
```

Convert Graph Data to Directed Graph

Using the table social_graphdata.Nodes, access the name property for each node that appears in the NodeData variable of the table.

Assign the table social_graphdata.Nodes to nodetable.

```
nodetable = social_graphdata.Nodes
```

```
nodetable=7×3 table
```

	NodeLabels	NodeData	NodeObject
0	'Person'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
1	'Person'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
2	'Person'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
3	'Person'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
4	'Person'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
5	'Person'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]
9	'Person'	[1×1 struct]	[1×1 database.neo4j.http.Neo4jNode]

Assign the row names for each row in the table nodetable to rownames.

```
rownames = nodetable.Properties.RowNames
```

```
rownames = 7×1 cell array
    {'0'}
    {'1'}
    {'2'}
    {'3'}
    {'4'}
```

```
{'5'}
{'9'}
```

Access the `NodeData` variable from `nodestable` for each row. `nodedata` contains an array of structures.

```
nodedata = [nodestable.NodeData{rownames}]
nodedata = 1x7 struct array with fields:
    name
```

To retrieve the name field from each structure, index into the array. `nodenames` is a cell array of character vectors that contains node names.

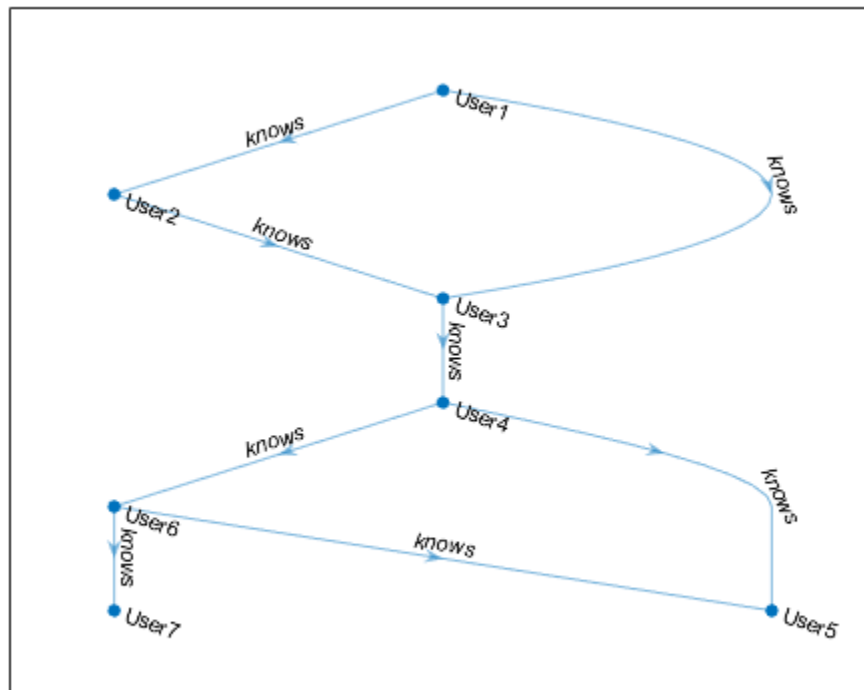
```
nodenames = {nodedata(:).name}
nodenames = 1x7 cell array
    {'User1'}    {'User3'}    {'User2'}    {'User4'}    {'User5'}    {'User6'}    {'User7'}
```

Create the `digraph` object `social_graph` using the `neo4jStruct2Digraph` function with the graph data stored in `social_graphdata` and the node names stored in `nodenames`.

```
social_graph = neo4jStruct2Digraph(social_graphdata, 'NodeNames', nodenames)
social_graph =
    digraph with properties:
        Edges: [8x3 table]
        Nodes: [7x3 table]
```

To see a visual representation of the graph, create a figure that displays `social_graph`.

```
plot(social_graph, 'EdgeLabel', social_graph.Edges.RelationType)
```



Find Shortest Path

Find the shortest path between User1 and User7 using `shortestpath`.

```
[user1_to_user7,distance] = shortestpath(social_graph, 'User1', 'User7')
```

```
user1_to_user7 = 1x5 cell array
    {'User1'}    {'User3'}    {'User4'}    {'User6'}    {'User7'}
```

```
distance = 4
```

Close Database Connection

```
close(neo4jconn)
```

See Also

`neo4j` | `searchNode` | `searchRelation` | `shortestpath`

Related Examples

- “Find Friends of Friends in Social Neighborhood” on page 9-32

More About

- “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

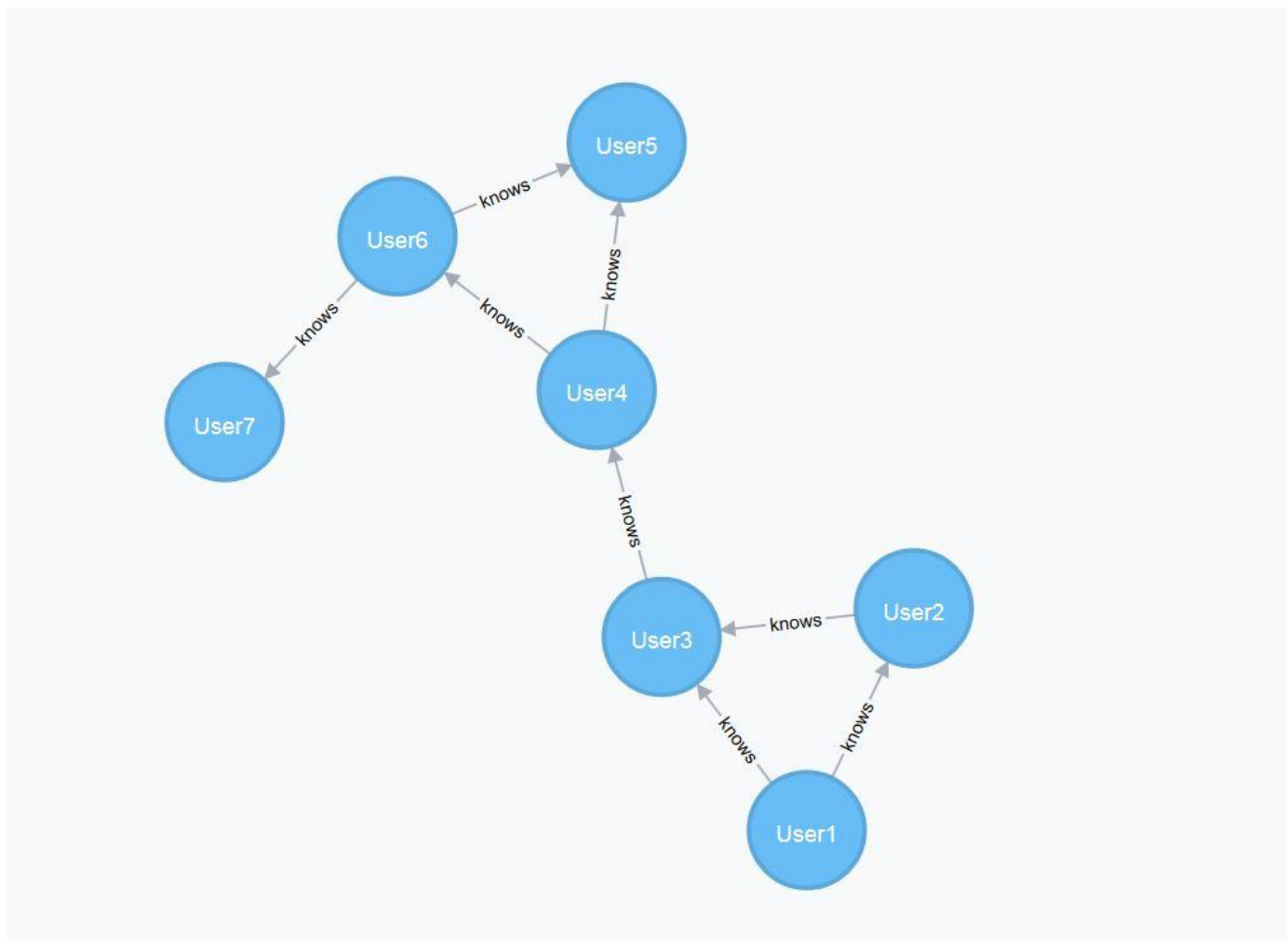
- “Directed and Undirected Graphs”

Find Friends of Friends in Social Neighborhood

This example shows how to search a social neighborhood to find the second-degree friends of a person, using the MATLAB® interface to Neo4j®. Assume that you have graph data that is stored on a Neo4j database which represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has type `knows`.

To find the second-degree friends of User1, use the MATLAB interface to Neo4j and the `digraph` object. For details about the MATLAB interface to Neo4j, see “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6.

The local machine hosts the Neo4j database with port number 7474, user name `neo4j`, and password `matlab`. For a visual representation of the data in the database, see this figure.



Connect to Neo4j Database

Create a Neo4j connection object `neo4jconn` using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Search for One Person in Database

Find the node that has the node label `Person` with the property name `User1`.

```
user1 = searchNode(neo4jconn, 'Person', 'PropertyKey', 'name', ...
    'PropertyValue', 'User1')

user1 =
    Neo4jNode with properties:
        NodeID: 0
        NodeData: [1x1 struct]
        NodeLabels: 'Person'
```

Search for All Second-Degree Friends of Person

Find outgoing relationships for `User1`. To limit the search to relationships with a distance of two or less, specify 2 as the value of the name-value pair argument `'Distance'`.

```
user1_relation = searchRelation(neo4jconn,user1,'out','Distance',2)

user1_relation = struct with fields:
    Origin: 0
    Nodes: [4x3 table]
    Relations: [4x5 table]
```

Convert Graph Data to Directed Graph

Using the table `user1_relation.Nodes`, access the `name` property for each node that appears in the `NodeData` variable of the table.

Assign the table `user1_relation.Nodes` to `nodetable`.

```
nodetable = user1_relation.Nodes
```

```
nodetable=4x3 table
    NodeLabels      NodeData      NodeObject
    _____      _____      _____
    0      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
    1      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
    2      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
```

```
3      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
```

Assign the row names for each row in the table `nodestable` to `rownames`.

```
rownames = nodestable.Properties.RowNames
```

```
rownames = 4x1 cell array
    {'0'}
    {'1'}
    {'2'}
    {'3'}
```

Access the `NodeData` variable from `nodestable` for each row. `nodedata` contains an array of structures.

```
nodedata = [nodestable.NodeData{rownames}]
```

```
nodedata = 1x4 struct array with fields:
    name
```

To retrieve the `name` field from each structure, index into the array. `nodenames` is a cell array of character vectors that contains node names.

```
nodenames = {nodedata(:).name}
```

```
nodenames = 1x4 cell array
    {'User1'}    {'User3'}    {'User2'}    {'User4'}
```

Create the `digraph` object `user1_graph` using the `neo4jStruct2Digraph` function with the relationship data stored in `user1_relation` and the node names stored in `nodenames`.

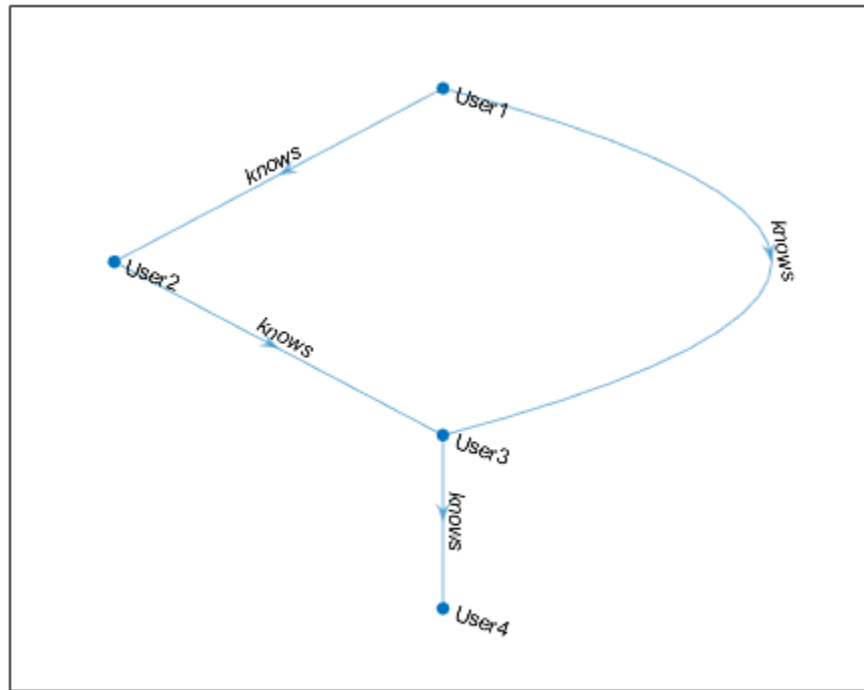
```
user1_graph = neo4jStruct2Digraph(user1_relation, 'NodeNames', nodenames)
```

```
user1_graph =
    digraph with properties:

    Edges: [4x3 table]
    Nodes: [4x3 table]
```

To see a visual representation of the graph, create a figure that displays `user1_graph`.

```
plot(user1_graph, 'EdgeLabel', user1_graph.Edges.RelationType)
```

Find Friends of Person

Retrieve a list of all the first-degree friends of User1. The `user1_friend` variable is a cell array of character vectors that contains the names of first-degree friends.

```

disp('Friends of User1 are:')

Friends of User1 are:

user1_friend = successors(user1_graph,'User1')

user1_friend = 2×1 cell array
    {'User3'}
    {'User2'}
  
```

Find Second-Degree Friends

To find the second-degree friends of User1, run `successors` again by looping through the list of the first-degree friends. `user1_friends_friend` is a cell array of character vectors that contains the names of second-degree friends.

```

user1_friends_friend = {};
for i = 1:length(user1_friend)
    user1_friends_friend = [user1_friends_friend; ...
        successors(user1_graph,user1_friend{i})];
end
disp('Friends of User1's friends are:')
  
```

Friends of User1's friends are:

```
user1_friends_friend = unique(user1_friends_friend)
user1_friends_friend = 2x1 cell array
    {'User3'}
    {'User4'}
```

Remove Duplicate Friends

Remove duplicates from the second-degree friends list that are already in the first-degree friends list using `setdiff`.

```
finalResult = setdiff(user1_friends_friend,user1_friend);
disp('User1's second-degree friends are:')
User1's second-degree friends are:
for i = 1:length(finalResult)
    disp(finalResult{i})
end
User4
```

`finalResult` is a cell array of character vectors that contains the names of second-degree friends. This list removes the names of the first-degree friends.

Close Database Connection

```
close(neo4jconn)
```

See Also

[neo4j](#) | [searchNode](#) | [searchRelation](#) | [successors](#) | [unique](#) | [setdiff](#)

Related Examples

- “Find Shortest Path Between People in Social Neighborhood” on page 9-27

More About

- “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6
- “Directed and Undirected Graphs”

Database Toolbox Interface for Neo4j Bolt Protocol Installation

To use the Database Toolbox Interface for Neo4j Bolt Protocol, you must first install it. Ensure that the interface supports your Neo4j database version.

Supported Neo4j Versions

The Database Toolbox Interface for Neo4j Bolt Protocol supports the Neo4j database version 4.0.0 and later.

Installation

To install the Database Toolbox Interface for Neo4j Bolt Protocol, follow these steps:

- 1 In the **Environment** section of the MATLAB toolstrip, select **Add-Ons > Get Add-Ons**.
- 2 In the Add-On Explorer, search for the Database Toolbox Interface for Neo4j Bolt Protocol.
- 3 Install the Database Toolbox Interface for Neo4j Bolt Protocol.
- 4 Restart MATLAB.

The installer downloads and installs the Neo4j Java driver. Then, the installer adds the driver to the static Java class path as part of the installation.

For details about installing add-ons, see “Get and Manage Add-Ons”. For other information, see “Add-Ons”.

See Also

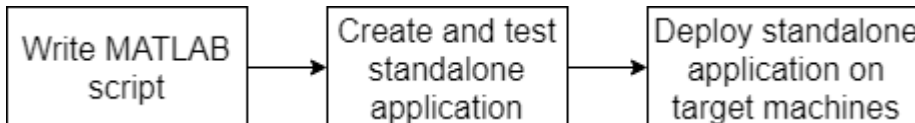
`neo4j` | `createNode` | `createRelation` | `storeDigraph` | `searchGraph` | `addNodeLabel` | `setNodeProperty` | `setRelationProperty`

More About

- “Find Friends of Friends in Social Neighborhood” on page 9-32
- “Update Friend Information in Social Neighborhood” on page 9-11
- “Search Graph Database” on page 9-9
- “Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

Deploy Graph Database Application with MATLAB Compiler

This example shows how to write a script to analyze data stored in a graph database, and deploy the script as a standalone application. Write code that connects to the Neo4j® database, imports data from the database into MATLAB®, analyzes the data, and closes the database connection. Then, you can deploy the code by compiling it as a standalone application by using the Application Compiler (MATLAB Compiler) app and running the application on other machines.



Overall, the example follows the steps described in “Create Standalone Application from MATLAB Function” (MATLAB Compiler) and updates the steps for a standalone database application.

Ensure that you have administrator privileges on the other machines to run the standalone application.

Create Function in MATLAB

Write a MATLAB script named `findShortestPathBetweenPeople.m` and save it in a file location of your choice. The script contains the `findShortestPathBetweenPeople` function, which returns the distance between two people in a graph network. The function performs these actions:

- Connects to a Neo4j database running on the local machine
- Imports graph data and converts it to a directed graph
- Performs the shortest path analysis
- Closes the database connection

```

type findShortestPathBetweenPeople.m

function distance = findShortestPathBetweenPeople(userA,userB)

% FINDSHORTESTPATHBETWEENPEOPLE The findShortestPathBetweenPeople function
% connects to a Neo4j® database, imports data from the database into
% MATLAB®, finds the shortest path between two people, and closes the
% database connection.

%%
% Create a Neo4j connection object |neo4jconn| using the URL
% |http://localhost:7474/db/data|, user name |neo4j|, and password
% |matlab|.

url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);

%%
% Find all the |Person| nodes and all the relationships associated with
% each |Person| node using |searchGraph|.
  
```

```

social_graphdata = searchGraph(neo4jconn,{'Person'});

%%
% Using the table |social_graphdata.Nodes|, access the |name| property for
% each node that appears in the |NodeData| variable of the table.
%
% Assign the table |social_graphdata.Nodes| to |nodestable|.

nodestable = social_graphdata.Nodes;

%%
% Assign the row names for each row in the table |nodestable| to
% |rownames|.

rownames = nodestable.Properties.RowNames;

%%
% Access the |NodeData| variable from |nodestable| for each row. |nodedata|
% contains an array of structures.

nodedata = [nodestable.NodeData{rownames}];

%%
% To retrieve the |name| field from each structure, index into the array.
% |nodenames| is a cell array of character vectors that contains node names.

nodenames = {nodedata(:).name};

%%
% Create the |digraph| object |social_graph| using the
% |neo4jStruct2Digraph| function with the graph data stored in
% |social_graphdata| and the node names stored in |nodenames|.

social_graph = neo4jStruct2Digraph(social_graphdata,'NodeNames',nodenames);

%%
% Find the shortest path between |UserA| and |UserB| using |shortestpath|.

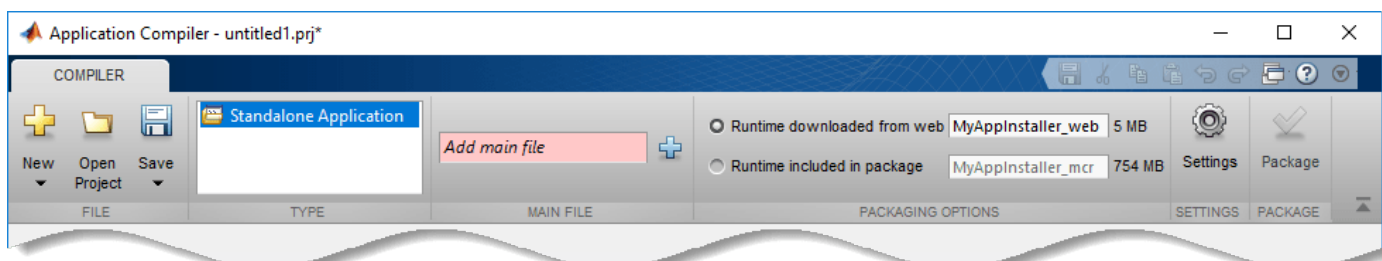
[~,distance] = shortestpath(social_graph,userA,userB);

%%
% Close the database connection.
close(neo4jconn)


```

Create Standalone Application Using Application Compiler App

On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow to open the apps gallery. Under **Application Deployment**, click **Application Compiler**.



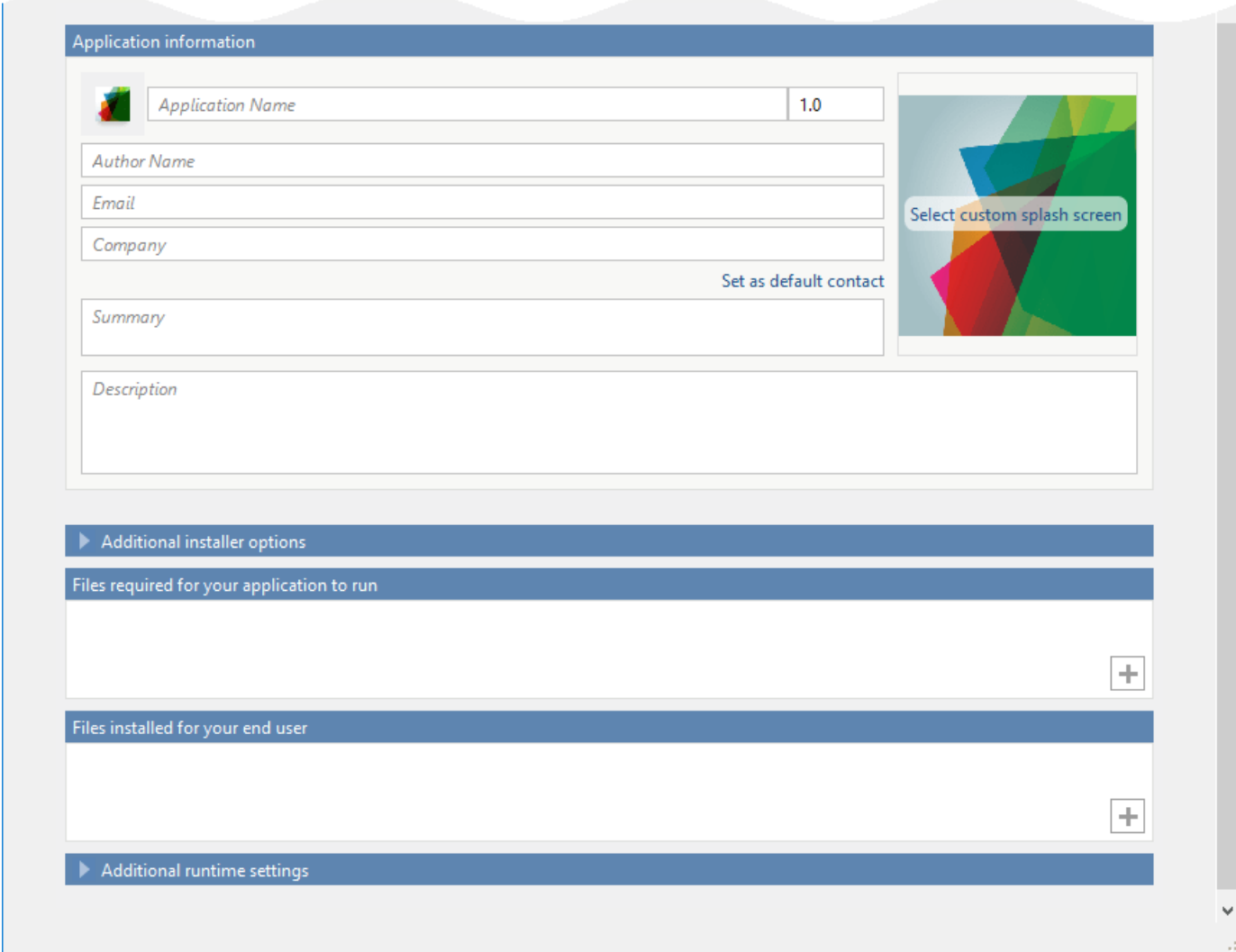
In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.

- 1 In the **Main File** section of the toolbar, click .
- 2 In the **Add Files** dialog box, browse to the file location that contains your saved script. Select `findShortestPathBetweenPeople.m` and click **Open**. The Application Compiler app adds the `findShortestPathBetweenPeople` function to the list of main files.

Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:

- **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application
- **Runtime included in package** — Generates an installer that includes the MATLAB Runtime installer

Customize the packaged application and its appearance by entering the following options:



The screenshot shows the 'Application information' section of the MATLAB Compiler packaging options dialog. It includes the following fields and options:

- Application Name:** A text field containing 'Application Name' and a version field set to '1.0'.
- Author Name:** A text field.
- Email:** A text field.
- Company:** A text field with a 'Set as default contact' link to its right.
- Summary:** A text area.
- Description:** A larger text area.
- Select custom splash screen:** A button with a colorful geometric background.
- Additional installer options:** A section header with a right-pointing arrow.
- Files required for your application to run:** A list area with a plus icon in the bottom right corner.
- Files installed for your end user:** A list area with a plus icon in the bottom right corner.
- Additional runtime settings:** A section header with a right-pointing arrow.

- **Application information** — Editable information about the deployed application. You can also customize the appearance of the standalone application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata.
- **Additional installer options** — Options for editing the default installation path for the generated installer and selecting a custom logo.
- **Files required for your application to run** — Additional files required by the generated application to run. The software includes these files in the generated application installer.
- **Files installed for your end user** — Files that are installed with your application. These files include the generated `readme.txt` file and the generated executable for the target platform.
- **Additional runtime settings** — Platform-specific options for controlling the generated executable.

For details about these options, see “Customize an Application” (MATLAB Compiler).

To generate the packaged application, click **Package** in the **Package** section on the toolbar. In the Save Project dialog box, specify the location in which to save the project.

In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output.

- `for_redistribution` — Folder containing the file that installs the application and the MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc` (such as binary, header, and source files for a specific target). Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler™.

Install and Run Standalone Application

To install the standalone application, in the `for_redistribution` folder, double-click the `MyAppInstaller_web` executable.

If you want to connect to the Internet using a proxy server, click **Connection Settings**. Enter the proxy server settings in the provided dialog box. Click **OK**.

To complete the installation, follow the instructions in the installation wizard.

To run your standalone application:

- 1 Open a terminal window.
- 2 Navigate to the folder in which you installed the application.
- 3 Run the application.

Test Standalone Application on Target Machine

Choose one target machine to test the MATLAB generated standalone application.

Copy the files in the `for_testing` folder to the target machine.

To test your standalone application:

- 1 Open a terminal window.
- 2 Navigate to the `for_testing` folder.
- 3 Run the application.

Deploy Standalone Application on Target Machines

Copy the `for_redistribution_files_only` folder to a file location on all target machines where MATLAB or MATLAB Runtime is installed and the Neo4j database server is running.

Run the MATLAB generated standalone application on all target machines by using the executable in the `for_redistribution_files_only` folder.

See Also

`neo4j` | `searchGraph` | `neo4jStruct2Digraph` | `close`

More About

- “Create Functions in Files”
- “Create Standalone Application from MATLAB Function” (MATLAB Compiler)
- “Customize an Application” (MATLAB Compiler)

MongoDB C++ Interface Topics

Import and Analyze Data from MongoDB Using MongoDB C++ Interface

This example shows how to import employee data from a collection in MongoDB® into the MATLAB® workspace using the MongoDB C++ interface. The example then shows how to conduct a simple data analysis based on the imported data.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server,port,dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [14x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 14 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Specify the `employees` collection for document retrieval. Retrieve all documents in the collection by using the MongoDB C++ interface connection. `documents` is a structure array.

```
collection = "employees";
documents = find(conn,collection);
```

Using all documents, determine the maximum salary of all employees. `salaries` contains an array of doubles for the salaries.

```
salaries = [];
for i = 1:length(documents)
```

```
        salaries = [salaries documents{i}.salary];  
end  
max(salaries)  
  
ans = int32  
      29000
```

Close the MongoDB connection.

```
close(conn)
```

See Also

[mongoc](#) | [isopen](#) | [find](#) | [close](#) | [max](#)

External Websites

- [MongoDB Manual](#)

Import Filtered Data from MongoDB Using MongoDB C++ Interface

This example shows how to import flight data from a MongoDB® collection into the MATLAB® workspace using the MongoDB C++ interface. The example then shows how to use a MongoDB query with filter criteria and a field list, and how to perform a simple data analysis based on the filtered flight data.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server,port,dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [14x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 14 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Specify the `airlinesmall` collection. Define the MongoDB query to filter the flight data for the years 1998 through 1999. Specify the fields to retrieve from the collection.

```
collection = "airlinesmall";
mongoquery = "{"Year":{"$gte":1998,"$lt":2000}}";
fields = strcat("{"Year":1.0,"Month":1.0,"DayOfMonth":1.0,"DayOfWeek":1.0," ...
    "DepTime":1.0,"ArrTime":1.0}");
```

Retrieve flight data using the MongoDB connection. `documents` is a structure array with fields that correspond to the specified fields.

```
documents = find(conn,collection,Query=mongoquery,Projection=fields)
```

```
documents=10911x1 struct array with fields:
```

```
  _id  
  Year  
  Month  
  DayofMonth  
  DayOfWeek  
  DepTime  
  ArrTime
```

Determine the unique years in the data.

```
years = [documents(:).Year];  
unique(years)
```

```
ans = 1x2 int32 row vector
```

```
    1998    1999
```

Close the MongoDB connection.

```
close(conn)
```

See Also

[mongoc](#) | [isopen](#) | [find](#) | [close](#) | [strcat](#) | [unique](#)

External Websites

- [MongoDB Manual](#)

Import Large Data from MongoDB Using MongoDB C++ Interface

This example shows how to import a large set of flight data from a MongoDB® collection into the MATLAB® workspace using the MongoDB C++ interface. To avoid out-of-memory issues when retrieving many documents, use a loop to import large data in batches.

Create MongoDB C++ Interface Connection

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server,port,dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [14x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 14 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Determine Number of Documents to Import

Find the total number of documents, specified as `totaldocs`, in the `airlinesmall` collection for the years 1997 through 2010. Use a MongoDB query to filter the flight data for the specified years.

```
collection = "airlinesmall";
mongoquery = {"Year":{"$gte":1997,"$lte":2010}};
totaldocs = count(conn,collection,Query=mongoquery);
```

Retrieve Large Data in Batches

Estimate the batch size to be 15,000 documents. Define the MATLAB workspace variable for storing the retrieved data.

```
batchsize = 15000;
flightdata = [];
```

You can change the batch size depending on the performance and memory capacity of your system.

Use a while loop to retrieve flight data from the collection. The variable `flightdata` accumulates each batch of retrieved data.

```
% Track number of documents read
index = 0;

while index < totaldocs

    % Retrieve documents in a batch
    localdata = find(conn, collection, Query=mongoquery, ...
        Skip=index, Limit=batchsize);

    % Store retrieved documents locally
    flightdata = [flightdata; localdata];

    % Move to the next batch
    index = index + batchsize;

end
```

Display information about the `flightdata` variable. The retrieved data is a structure array that contains 75,603 structures. Each structure contains 30 fields of flight data.

```
whos flightdata
```

Name	Size	Bytes	Class	Attributes
flightdata	75603x1	248848692	struct	

Close MongoDB C++ Interface Connection

```
close(conn)
```

See Also

`mongoc` | `isopen` | `count` | `find` | `close`

External Websites

- [MongoDB Manual](#)

Export MATLAB Data into MongoDB Using MongoDB C++ Interface

This example shows how to export table and structure data from the MATLAB® workspace into new MongoDB® collections using the MongoDB C++ interface. The example then shows how to count the number of documents in the collections, remove documents from the collections, and drop the collections.

The example uses the data set `tsunamis.xlsx`, which contains tsunami data. You can find this file in the `toolbox/matlab/demos` folder.

Create MongoDB C++ Interface Connection

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server,port,dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Create Collection and Export Data into MongoDB

Load the data set using the `readtable` function. Convert tsunami data to a structure using the `table2struct` function. The MATLAB workspace contains the `tsunamidata` structure.


```
data = readtable("tsunamis.xlsx");  
tsunamidata = table2struct(data);
```

Create a collection to store the tsunami data using the MongoDB connection.

```
tsunamicoll = "tsunamis";  
createCollection(conn,tsunamicoll)
```

Export structure data into the `tsunamis` collection. `n` contains the number of documents inserted.

```
n = insert(conn,tsunamicoll,tsunamidata)
```

```
n = int64  
    162
```

Count Documents in Collection

Count the number of documents in the new collection.

```
ntsunamis = count(conn,tsunamicoll)
```

```
ntsunamis = int64  
    162
```

Remove Documents and Drop Collection

Remove all documents from the collection. `ntsunamis` contains the number of documents removed from the collection.

```
ntsunamis = remove(conn,tsunamicoll,"{}")
```

```
ntsunamis = int64  
    162
```

Drop the collection from the `mongotest` database.

```
dropCollection(conn,tsunamicoll)
```

Close MongoDB C++ Interface Connection

```
close(conn)
```

See Also

`mongoc` | `isopen` | `count` | `createCollection` | `dropCollection` | `insert` | `remove` | `close` | `readtable`

External Websites

- [MongoDB Manual](#)

Import and Export MATLAB Objects Using MongoDB and MongoDB C++ Interface

This example shows how to export objects from the MATLAB workspace into MongoDB using the MongoDB C++ interface. The export serializes the objects in MongoDB. Then, the example shows how to import objects back into the MATLAB workspace. The import deserializes the objects and recreates them in MATLAB for method execution. After the export and import, the example shows how to drop the collection.

In this example, the objects belong to the `TensileData` class. This class is a sample class in MATLAB. The data used to create the objects is sample data. For details, see “Representing Structured Data with Classes”. To run the code in this example, you define the class in the current folder.

The sample data represents tensile stress and strain measurements that you can use to calculate the elastic modulus of various materials. In simple terms, stress is the force applied to a material, and strain is the resulting deformation. The ratio of stress to strain defines a characteristic of the material.

Create Objects

Create the `TensileData` objects `tdcs` for carbon steel materials and `tdss` for stainless steel materials.

```
tdcs = TensileData('carbon steel',1, ...
    [2e4 4e4 6e4 8e4],[.12 .20 .31 .40]);
tdss = TensileData('stainless steel',1, ...
    [2e4 4e4 6e4 8e4],[.06 .10 .16 .20]);
```

Connect to MongoDB C++ Interface

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server,port,dbname)
```

```
conn =
  connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `mongo` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is `27017`.

- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)
```

```
ans =
```

```
    logical
```

```
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Create Collection in MongoDB

Create the `TensileData` collection using the MongoDB connection.

```
collection = "TensileData";
createCollection(conn, collection)
```

Export Objects into MongoDB

Export the `TensileData` objects into the collection. The `insert` function serializes the `TensileData` objects into a JSON-style structure. `ntdcs` and `ntdss` contain the number of objects exported into the collection.

```
ntdcs = insert(conn, collection, tdc);
ntdss = insert(conn, collection, tdss);
```

Import Objects into MATLAB Workspace

Import the `TensileData` objects into the MATLAB workspace. The `find` function deserializes the `TensileData` objects into the `documents` structure array.

```
documents = find(conn, collection);
```

Recreate the objects in the MATLAB workspace.

```
tdcs = TensileData(documents(1).Material, documents(1).SampleNumber, ...
    documents(1).Stress, documents(1).Strain);
tdss = TensileData(documents(2).Material, documents(2).SampleNumber, ...
    documents(2).Stress, documents(2).Strain);
```

You can execute methods of the objects after they appear in the MATLAB workspace. For example, calculate the elastic modulus.

Remove Documents and Drop Collection

Remove all documents from the collection. `n` contains the number of documents removed from the collection.

```
n = remove(conn, collection, "{}")
```

```
n =
```

```
     2
```

Drop the collection.

```
dropCollection(conn, collection)
```

Close MongoDB C++ Interface Connection

```
close(conn)
```

See Also

[mongoc](#) | [isopen](#) | [find](#) | [createCollection](#) | [dropCollection](#) | [insert](#) | [remove](#) | [close](#)

External Websites

- [MongoDB Manual](#)

Functions

close

Package: database.odbc

Close and invalidate database and driver resource utilizer

Syntax

```
close(object)
```

Description

`close(object)` closes and invalidates the database and driver resource utilizer `object` to free up database and driver resources.

Examples

Close connection Object

Connect to a Microsoft® SQL Server® database and verify the database connection. Then, import data from the database into MATLAB®. Determine the highest unit cost among the retrieved products in the table. Close the database connection.

Create an ODBC database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Select all data from `productTable` and sort it by the product number. `data` is a table containing the imported data that results from executing the SQL `SELECT` statement.

```
selectquery = 'SELECT * FROM productTable ORDER BY productNumber';  
data = select(conn,selectquery);
```

Display the first three rows of `data`.

```
data(1:3, :)
```

```
ans =
```

3x5 table

productNumber	stockNumber	supplierNumber	unitCost	productDescription
1	4.0035e+05	1001	14	'Building Blocks'
2	4.0031e+05	1002	9	'Painting Set'
3	4.01e+05	1009	17	'Slinky'

Determine the highest unit cost in the table.

```
max(data.unitCost)
```

```
ans =
```

```
24
```

Close the database connection.

```
close(conn)
```

Close DatabaseDatastore Object

Create a database connection using a JDBC driver. To create this connection, you must configure a JDBC data source. For more information, see the `configureJDBCDataSource` function. Then, create a `DatabaseDatastore` object and close it.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
username = "";
password = "";
conn = database(datasource,username,password);
```

Create a `DatabaseDatastore` object using the database connection and an SQL query. This SQL query retrieves all data from the `airlinesmall` table.

```
sqlquery = "select * from airlinesmall";
dbds = databaseDatastore(conn,sqlquery);
```

Close the `DatabaseDatastore` object and the database connection.

```
close(dbds)
```

Input Arguments

object — Database and driver resource utilizer

connection object | `DatabaseDatastore` object

Database and driver resource utilizer, specified as one of the objects described in this table.

Object Argument Name	Object Name	Object Description	Object Creation Function
conn	connection	Create a connection between an installed database and MATLAB. For details, see “Connect to Database” on page 2-121.	database
dbds	DatabaseDatastore	Create a connection to a type of datastore for working with large data.	databaseDatastore

- connection objects and DatabaseDatastore objects remain open until you close them using the `close` function. Always close these objects when you finish using them.
- Executing `close` with a DatabaseDatastore object releases the MATLAB resources associated with the connection object.

Note When you close the MATLAB session, MATLAB closes open DatabaseDatastore objects and connections. However, the database might not free up the connections. Consult your database administrator about the remaining connections.

Version History

Introduced before R2006a

See Also

database | databaseDatastore | fetch

Topics

“Import Data from Database Table Using `sqlread` Function” on page 5-58

“Import Large Data Using DatabaseDatastore Object” on page 5-23

“Insert Data into Database Table” on page 5-61

“Configure Driver and Data Source” on page 2-14

“Connect to Database” on page 2-121

commit

Package: database.odbc

Make database changes permanent

Syntax

```
commit(conn)
```

Description

`commit(conn)` makes permanent changes made to the database connection `conn` since the last `commit` or `rollback` function was run. To run this function, the `AutoCommit` flag for `conn` must be off.

Examples

Example 1 — Check the Status of the Autocommit Flag

Check that the status of the `AutoCommit` flag for connection `conn` is off.

```
conn.AutoCommit  
ans =  
    'off'
```

Example 2 — Commit Data to a Database

- 1 Insert `exdata` into the columns `DEPTNO`, `DNAME`, and `LOC` in the table `DEPT`, for the data source `conn`.

```
datainsert(conn, 'DEPT', ...  
{ 'DEPTNO'; 'DNAME'; 'LOC' }, exdata)
```

- 2 Commit this data.

```
commit(conn)
```

Tips

For ODBC connections, you can use the `commit` function with the native ODBC interface. For details, see `database`.

Version History

Introduced before R2006a

See Also

`database` | `exec` | `datainsert` | `get` | `rollback` | `update`

Topics

“Import Data from Database Table Using sqlread Function” on page 5-58

“Insert Data into Database Table” on page 5-61

“Roll Back Data After Updating Record” on page 5-9

commit

Make changes to SQLite database file permanent

Syntax

```
commit(conn)
```

Description

`commit(conn)` makes changes to the SQLite database connection permanent using the MATLAB interface to SQLite. This function makes permanent any changes made after the last `commit` or `rollback` function has been run. To use the `commit` function, you must set the `AutoCommit` property of the `sqlite` object to `off`.

Examples

Commit Data to SQLite Database

Use the MATLAB® interface to SQLite to insert product data from MATLAB into a new table in an SQLite database. Then, commit the changes to the database.

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Allow manual committing of changes to the database by setting the `AutoCommit` property to `off`.

```
conn.AutoCommit = "off";
```

Create a MATLAB table that contains data for two products. The data is stored in the `productTable` and `suppliers` tables.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"],'VariableNames',["productNumber" ...
    "stockNumber" "supplierNumber" "unitCost" "productDescription"]);
```

Insert the product data into a new table named `toyTable`.

```
tablename = "toyTable";
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
    productNumber    stockNumber    supplierNumber    unitCost    productDescription
```

30	5e+05	1000	25	"Rubik's Cube"
40	6e+05	2000	30	"Doll House"

Commit the changes to the database.

```
commit(conn)
```

Delete the new table to maintain the dataset.

```
sqlquery = "DROP TABLE toyTable";  
execute(conn, sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

Version History

Introduced in R2022a

See Also

Objects

sqlite

Functions

sqlread | sqlwrite | close

Topics

"Insert Data into SQLite Database Table" on page 5-36

"Create Table and Add Column in SQLite Database" on page 5-38

"Delete Data from SQLite Database" on page 5-39

"Roll Back Data in SQLite Database" on page 5-41

configureODBCDataSource

Open ODBC Data Source Administrator dialog box

Syntax

```
configureODBCDataSource
```

Description

configureODBCDataSource opens the ODBC Data Source Administrator dialog box on Windows systems.

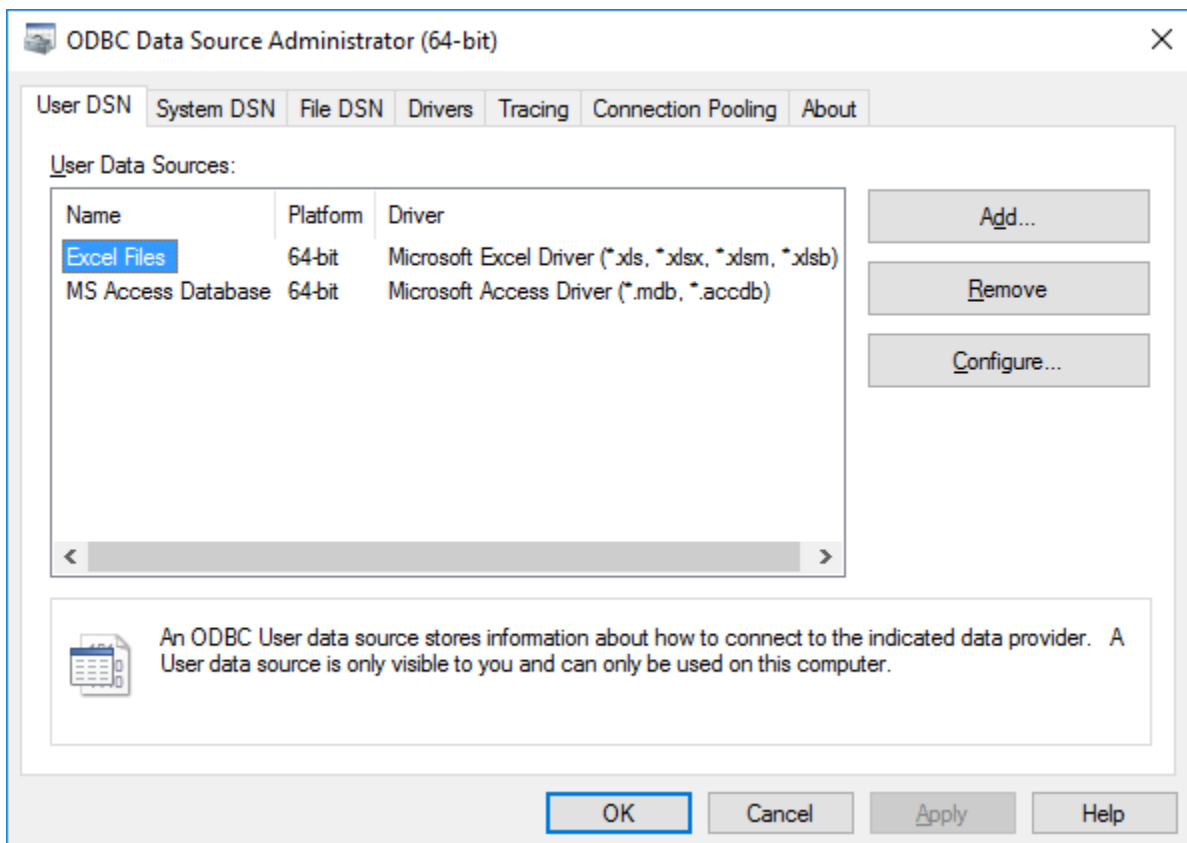
Examples

Open ODBC Data Source Administrator Dialog Box

Use the configureODBCDataSource function to open the ODBC Data Source Administrator dialog box on Windows systems.

```
configureODBCDataSource
```

The ODBC Data Source Administrator dialog box opens.



For configuring ODBC data sources, see “Configure Driver and Data Source” on page 2-14.

Tips

- Database Toolbox no longer supports connecting to a database using a 32-bit driver. For Microsoft Access, use the 64-bit version. Or, to connect to the 32-bit version of Microsoft Access, see <https://www.mathworks.com/matlabcentral/answers/235949-how-to-connect-to-32-bit-microsoft-access-database-from-64-bit-matlab>. For details about working with the 64-bit version of Windows, see Using Previous MATLAB Releases.

Alternative Functionality

App

You can open the ODBC Data Source Administrator dialog box using the **Database Explorer** app. In the **Data Source** section of the **Database Explorer** app, select **Configure Data Source > Configure ODBC data source**.

Version History

Introduced in R2017b

See Also

Apps

Database Explorer

Functions

database

Topics

“Configure Driver and Data Source” on page 2-14

configureJDBCDataSource

(To be removed) Configure JDBC data source

Note The `configureJDBCDataSource` function will be removed in a future release. Use the `databaseConnectionOptions` function instead. For details, see “Compatibility Considerations”.

Syntax

```
configureJDBCDataSource
opts = configureJDBCDataSource('Vendor', vendor)
opts = configureJDBCDataSource('DataSource', datasource)
```

Description

`configureJDBCDataSource` opens the JDBC Data Source Configuration dialog box.

`opts = configureJDBCDataSource('Vendor', vendor)` creates a new JDBC data source for the specified database vendor.

`opts = configureJDBCDataSource('DataSource', datasource)` edits an existing JDBC data source.

Examples

Open JDBC Data Source Configuration Dialog Box

Use the `configureJDBCDataSource` function to open the JDBC Data Source Configuration dialog box.

```
configureJDBCDataSource
```

The JDBC Data Source Configuration dialog box opens.

JDBC Data Source Configuration

Data Source Details

Name:

Vendor: Microsoft SQL Server
MySQL
Oracle
PostgreSQL

Driver Location: ...

Connection Parameters

Database:

Server:

Port Number:

Authentication: Server ▼

Connection Options

+ -

	Name	Value
1		
2		
3		

Edit Test Save Delete

Message

For information about configuring JDBC data sources, see “Configure Driver and Data Source” on page 2-14.

Create JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft SQL Server database.

Create an SQL Server data source.

```
opts = configureJDBCDataSource('Vendor', 'Microsoft SQL Server')
```

```
opts =
```

```
  JDBCConnectionOptions with properties:
      Vendor: 'Microsoft SQL Server'
  DataSourceName: ''
      DatabaseName: ''
      Server: 'localhost'
      PortNumber: 1433
      AuthType: 'Server'
  JDBCDriverLocation: ''
```

opts is a JDBCConnectionOptions object with these properties:

- Vendor — Database vendor name
- DataSourceName — Name of the data source
- DatabaseName — Name of the database
- Server — Name of the database server
- PortNumber — Port number
- AuthType — Authentication type
- JDBCDriverLocation — Full path of the JDBC driver file

Configure the data source by setting the JDBC connection options for the data source SQLServerDataSource, database server dbtb04, port number 54317, full path to the JDBC driver file, and Windows® authentication.

```
opts = setConnectionOptions(opts, ...
  'DataSourceName', 'SQLServerDataSource', ...
  'Server', 'dbtb04', 'PortNumber', 54317, ...
  'JDBCDriverLocation', 'C:\Drivers\sqljdbc4.jar', ...
  'AuthType', 'Windows')
```

```
opts =
```

```
  JDBCConnectionOptions with properties:
      Vendor: 'Microsoft SQL Server'
  DataSourceName: 'SQLServerDataSource'
      DatabaseName: ''
      Server: 'dbtb04'
      PortNumber: 54317
      AuthType: 'Windows'
  JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

The `setConnectionOptions` function sets the `DataSourceName`, `Server`, `PortNumber`, `AuthType`, and `JDBCDriverLocation` properties in the `JDBCConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";  
password = "";  
status = testConnection(opts,username,password)
```

```
status = logical
```

```
1
```

Save the configured data source.

```
saveAsJDBCDataSource(opts)
```

You can connect to the new data source using the `database` function or the Database Explorer app.

Input Arguments

vendor — Database vendor

'Microsoft SQL Server' | 'MySQL' | 'Oracle' | 'PostgreSQL' | 'Other'

Database vendor, specified as one of these values:

- 'Microsoft SQL Server' — Microsoft SQL Server database
- 'MySQL' — MySQL database
- 'Oracle' — Oracle database
- 'PostgreSQL' — PostgreSQL database
- 'Other' — Other database

You can specify these values as either a character vector or string scalar.

datasource — Data source name

character vector | string scalar

Data source name, specified as a character vector or string scalar. Specify the name of an existing configured JDBC data source.

Example: "myJDBCDataSource"

Data Types: char | string

Output Arguments

opts — JDBC connection options

JDBCConnectionOptions object

JDBC connection options, returned as a `JDBCConnectionOptions` object.

Alternative Functionality

App

You can open the JDBC Data Source Configuration dialog box using the **Database Explorer** app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure JDBC data source**.

Version History

Introduced in R2019a

R2020b: configureJDBCDataSource function will be removed

Not recommended starting in R2020b

The configureJDBCDataSource function will be removed in a future release. Use the databaseConnectionOptions function instead. Some differences between the workflows might require updates to your code.

Update Code

Use the databaseConnectionOptions function to create an SqlConnectionOptions object.

In prior releases, you configured a JDBC data source using the configureJDBCDataSource function and the JDBCConnectionOptions object. For example:

```
opts = configureJDBCDataSource('Vendor','Microsoft SQL Server');
opts = setConnectionOptions(opts, ...
    'DataSourceName','SQLServerDataSource', ...
    'Server','dbtb04','PortNumber',54317, ...
    'JDBCDriverLocation','C:\Drivers\sqljdbc4.jar', ...
    'AuthType','Windows');
username = '';
password = '';
status = testConnection(opts,username,password);
saveAsJDBCDataSource(opts)
```

Now you can set JDBC connection options using the databaseConnectionOptions function, and save the data source using the SqlConnectionOptions object instead.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc",vendor);
opts = setoptions(opts, ...
    'DataSourceName',"SQLServerDataSource", ...
    'JDBCDriverLocation',"C:\Drivers\sqljdbc4.jar", ...
    'DatabaseName',"toystore_doc",'Server',"dbtb04", ...
    'PortNumber',54317,'AuthType',"Windows");
username = "";
password = "";
status = testConnection(opts,username,password);
saveAsDataSource(opts)
```

Also, you can open the JDBC Data Source Configuration dialog box by using the **Database Explorer** app.

See Also

Apps

Database Explorer

Objects

JDBCConnectionOptions

Functions

setConnectionOptions | addConnectionOptions | rmConnectionOptions | testConnection
| saveAsJDBCDataSource | deleteJDBCDataSource | database |
databaseConnectionOptions

Topics

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

JDBCConnectionOptions

(To be removed) Define JDBC connection options for database

Note The JDBCConnectionOptions object will be removed in a future release. Use the SQLConnectionOptions object instead. For details, see “Compatibility Considerations”.

Description

The JDBCConnectionOptions object enables you to configure a JDBC data source and set JDBC connection options.

Creation

Create a JDBCConnectionOptions object with the configureJDBCDataSource function.

Properties

All Databases

JBCDriverLocation — JDBC driver location

character vector | string scalar

JDBC driver location, specified as a character vector or string scalar. Specify the full path to the JDBC driver file, including the name of the file.

Example: "C:\drivers\sqljdbc4.jar"

Data Types: char | string

Vendor — Database vendor

character vector

This property is read-only.

Database vendor, specified as a character vector.

Example: 'MySQL'

Data Types: char

DataSourceName — Data source name

character vector | string scalar

Data source name, specified as a character vector or string scalar.

Example: "mydatasource"

Data Types: char | string

Common Properties for Microsoft SQL Server, MySQL, Oracle, and PostgreSQL Databases**Server — Database server name or address**`'localhost'` (default) | character vector | string scalar

Database server name or address, specified as a character vector or string scalar.

Data Types: `char` | `string`

PortNumber — Server port number where the server is listening

numeric scalar

Server port number where the server is listening, specified as a numeric scalar. The default value is based on the database vendor:

- Microsoft SQL Server — 1433
- MySQL — 3306
- Oracle — 1521
- PostgreSQL — 5432

Data Types: `double`

DatabaseName — Database name

character vector | string scalar

Database name on the server, specified as a character vector or string scalar.

Example: `"mydatabase"`

Data Types: `char` | `string`

Microsoft SQL Server Database Only**AuthType — Authentication type**`'Server'` (default) | `'Windows'`

Authentication type, specified as one of these values:

- `'Server'` — Microsoft SQL Server authentication
- `'Windows'` — Windows authentication

You can specify these values as either a character vector or string scalar.

Oracle Database Only**DriverType — Driver type**`'thin'` (default) | `'oci'`

Driver type, specified as one of these values:

- `'thin'` — Thin driver
- `'oci'` — Windows authentication

You can specify these values as either a character vector or string scalar.

Other Databases

Driver — JDBC driver name

character vector | string scalar

JDBC driver name, specified as a character vector or string scalar that refers to the Java driver that implements the `java.sql.Driver` interface.

For details about the JDBC driver name, consult your database driver documentation.

Data Types: `char` | `string`

URL — Database connection URL

character vector | string scalar

Database connection URL, specified as a character vector or string scalar for the vendor-specific URL. This URL is typically constructed using connection properties such as the server name, port number, and database name.

For details about the database connection URL, consult your database driver documentation.

Data Types: `char` | `string`

Object Functions

<code>setConnectionOptions</code>	(To be removed) Set JDBC connection options
<code>addConnectionOptions</code>	(To be removed) Add JDBC driver-specific connection options
<code>rmConnectionOptions</code>	(To be removed) Remove JDBC driver-specific connection options
<code>testConnection</code>	(To be removed) Test JDBC data source connection
<code>saveAsJDBCDataSource</code>	(To be removed) Save JDBC data source

Examples

Create JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft SQL Server database.

Create an SQL Server data source.

```
opts = configureJDBCDataSource('Vendor', 'Microsoft SQL Server')
```

```
opts =
```

```
JDBCConnectionOptions with properties:
```

```

      Vendor: 'Microsoft SQL Server'
DataSourceName: ''

      DatabaseName: ''
      Server: 'localhost'
      PortNumber: 1433
      AuthType: 'Server'
```

```
JBCDDriverLocation: ''
```

`opts` is a `JDBCConnectionOptions` object with these properties:

- `Vendor` — Database vendor name
- `DataSourceName` — Name of the data source
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number
- `AuthType` — Authentication type
- `JDBCDriverLocation` — Full path of the JDBC driver file

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, database server `dbtb04`, port number `54317`, full path to the JDBC driver file, and Windows® authentication.

```
opts = setConnectionOptions(opts, ...
    'DataSourceName', 'SQLServerDataSource', ...
    'Server', 'dbtb04', 'PortNumber', 54317, ...
    'JDBCDriverLocation', 'C:\Drivers\sqljdbc4.jar', ...
    'AuthType', 'Windows')
```

```
opts =
```

```
    JDBCConnectionOptions with properties:
```

```
        Vendor: 'Microsoft SQL Server'
DataSourceName: 'SQLServerDataSource'

        DatabaseName: ''
           Server: 'dbtb04'
      PortNumber: 54317
         AuthType: 'Windows'
```

```
        JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

The `setConnectionOptions` function sets the `DataSourceName`, `Server`, `PortNumber`, `AuthType`, and `JDBCDriverLocation` properties in the `JDBCConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)
```

```
status = logical
```

```
    1
```

Save the configured data source.

```
saveAsJDBCDataSource(opts)
```


You can connect to the new data source using the database function or the Database Explorer app.

Version History

Introduced in R2019b

R2020b: JDBCConnectionOptions object will be removed

Not recommended starting in R2020b

The JDBCConnectionOptions object will be removed in a future release. Use the SqlConnectionOptions object instead. Some differences between the workflows might require updates to your code.

Update Code

Use the SqlConnectionOptions object to set JDBC connection options.

In prior releases, you configured a JDBC data source using the JDBCConnectionOptions object. For example:

```
opts = configureJDBCDataSource('Vendor','Microsoft SQL Server');
opts = setConnectionOptions(opts, ...
    'DataSourceName','SQLServerDataSource', ...
    'Server','dbtb04','PortNumber',54317, ...
    'JDBCDriverLocation','C:\Drivers\sqljdbc4.jar', ...
    'AuthType','Windows');
username = "";
password = "";
status = testConnection(opts,username,password);
saveAsJDBCDataSource(opts)
```

Now you can set JDBC connection options and save the data source using the SqlConnectionOptions object instead.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc",vendor);
opts = setoptions(opts, ...
    'DataSourceName',"SQLServerDataSource", ...
    'JDBCDriverLocation',"C:\Drivers\sqljdbc4.jar", ...
    'DatabaseName',"toystore_doc",'Server',"dbtb04", ...
    'PortNumber',54317,'AuthType',"Windows");
username = "";
password = "";
status = testConnection(opts,username,password);
saveAsDataSource(opts)
```

See Also

Objects

SqlConnectionOptions

Functions

configureJDBCDataSource | deleteJDBCDataSource

Topics

"Modify and Delete Data Sources" on page 4-17

“Configure Driver and Data Source” on page 2-14

addConnectionOptions

Package: database.options

(To be removed) Add JDBC driver-specific connection options

Note The addConnectionOptions function will be removed in a future release. Use the setoptions function instead. For details, see “Compatibility Considerations”.

Syntax

```
opts = addConnectionOptions(opts,
Option1,OptionValue1,...,OptionN,OptionValueN)
```

Description

opts = addConnectionOptions(opts, Option1,OptionValue1,...,OptionN,OptionValueN) adds JDBC driver-specific connection options using the JDBCConnectionOptions object opts.

Examples

Add Driver-Specific Connection Option

Create a JDBC data source for a Microsoft SQL Server database, configure the data source by setting JDBC connection options, set an additional JDBC driver-specific option, and save the data source.

Create an SQL Server data source.

```
opts = configureJDBCDataSource('Vendor','Microsoft SQL Server')
```

```
opts =
```

```
    JDBCConnectionOptions with properties:
```

```
        Vendor: 'Microsoft SQL Server'
DataSourceName: ''
```

```
        DatabaseName: ''
           Server: 'localhost'
        PortNumber: 1433
           AuthType: 'Server'
```

```
JDBCDriverLocation: ''
```

opts is a JDBCConnectionOptions object with these properties:

- Vendor — Database vendor name

- `DataSourceName` — Name of the data source
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number
- `AuthType` — Authentication type
- `JDBCDriverLocation` — Full path of the JDBC driver file

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, database server `dbtb04`, port number `54317`, full path to the JDBC driver file, and Windows® authentication.

```
opts = setConnectionOptions(opts, ...
    'DataSourceName', 'SQLServerDataSource', ...
    'Server', 'dbtb04', 'PortNumber', 54317, ...
    'JDBCDriverLocation', 'C:\Drivers\sqljdbc4.jar', ...
    'AuthType', 'Windows')
```

opts =

JDBCConnectionOptions with properties:

```
Vendor: 'Microsoft SQL Server'
DataSourceName: 'SQLServerDataSource'
```

```
DatabaseName: ''
Server: 'dbtb04'
PortNumber: 54317
AuthType: 'Windows'
```

```
JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

The `setConnectionOptions` function sets the `DataSourceName`, `Server`, `PortNumber`, `AuthType`, and `JDBCDriverLocation` properties in the `JDBCConnectionOptions` object.

Add a JDBC driver-specific connection option by using a name-value pair argument. The option specifies a timeout value for establishing the database connection. `opts` contains a new section of properties for the additional JDBC connection option.

```
opts = addConnectionOptions(opts, 'loginTimeout', 20)
```

opts =

JDBCConnectionOptions with properties:

```
Vendor: 'Microsoft SQL Server'
DataSourceName: 'SQLServerDataSource'
```

```
DatabaseName: ''
Server: 'dbtb04'
PortNumber: 54317
AuthType: 'Windows'
```

```
JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

Additional JDBC Connection Options:

```
loginTimeout: '20'
```

Test the database connection with a blank user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts,username,password)
```

```
status = logical
```

```
1
```

Save the configured data source.

```
saveAsJDBCDataSource(opts)
```

Input Arguments

opts — JDBC connection options

JDBCConnectionOptions object

JDBC connection options, specified as a JDBCConnectionOptions object.

Option1,OptionValue1, . . . ,OptionN,OptionValueN — JDBC driver-specific options

name-value pair arguments

JDBC driver-specific options, specified as one or more name-value pair arguments. `Option` is a character vector or string scalar that specifies the name of a JDBC driver-specific connection option. `OptionValue` specifies the value of the connection option. `OptionValue` can be a character vector, string scalar, logical scalar, or numeric scalar. You can specify any connection option allowed by your JDBC driver. Consult your JDBC driver documentation for the available connection options.

Note You can specify a JDBC driver-specific connection option in two ways: using this input argument or using the database connection URL. If you specify the same option with different values in both the input argument and URL, the JDBC driver implementation defines which value takes precedence. For maximum portability, specify a JDBC driver-specific connection option in only one of these ways.

Example: 'useSSL', true specifies that the database connection uses SSL authentication.

Output Arguments

opts — JDBC connection options

JDBCConnectionOptions object

JDBC connection options, returned as a JDBCConnectionOptions object.

Alternative Functionality

App

You can add JDBC driver-specific connection options by using the JDBC Data Source Configuration dialog box in the **Database Explorer** app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure JDBC data source**.

Version History

Introduced in R2019b

R2020b: addConnectionOptions function will be removed

Not recommended starting in R2020b

The addConnectionOptions function will be removed in a future release. Use the setoptions function instead. Some differences between the workflows might require updates to your code.

Update Code

Use the setoptions function with the SqlConnectionOptions object to set JDBC driver-specific connection options.

In prior releases, you configured a JDBC data source using the JDBCConnectionOptions object, and added options using the addConnectionOptions function. For example:

```
opts = configureJDBCDataSource('Vendor','Microsoft SQL Server');
opts = setConnectionOptions(opts, ...
    'DataSourceName','SQLServerDataSource', ...
    'Server','dbtb04','PortNumber',54317, ...
    'JDBCDriverLocation','C:\Drivers\sqljdbc4.jar', ...
    'AuthType','Windows');
opts = addConnectionOptions(opts,'loginTimeout',20);
username = "";
password = "";
status = testConnection(opts,username,password);
saveAsJDBCDataSource(opts)
```

Now you can set JDBC driver-specific connection options using the setoptions function with the SqlConnectionOptions object instead.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc",vendor);
opts = setoptions(opts, ...
    'DataSourceName',"SQLServerDataSource", ...
    'JDBCDriverLocation',"C:\Drivers\sqljdbc4.jar", ...
    'DatabaseName',"toystore_doc",'Server',"dbtb04", ...
    'PortNumber',54317,'AuthType',"Windows", ...
    'loginTimeout',20);
username = "";
password = "";
status = testConnection(opts,username,password);
saveAsDataSource(opts)
```

See Also

Objects

JDBCConnectionOptions

Functions

configureJDBCDataSource | saveAsJDBCDataSource | setConnectionOptions |
testConnection | rmConnectionOptions | deleteJDBCDataSource | setoptions

Topics

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

deleteJDBCDataSource

(To be removed) Delete JDBC data source

Note The `deleteJDBCDataSource` function will be removed in a future release. Use the `deleteDataSource` function instead.

Syntax

```
deleteJDBCDataSource(datasource)
```

Description

`deleteJDBCDataSource(datasource)` deletes the specified JDBC data source.

Examples

Delete JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft SQL Server database. Then, delete the configured data source.

Create an SQL Server data source.

```
opts = configureJDBCDataSource('Vendor', 'Microsoft SQL Server')
```

```
opts =
```

```
  JDBCConnectionOptions with properties:
```

```
      Vendor: 'Microsoft SQL Server'  
DataSourceName: ''
```

```
  DatabaseName: ''  
      Server: 'localhost'  
  PortNumber: 1433  
      AuthType: 'Server'
```

```
  JDBCDriverLocation: ''
```

`opts` is a `JDBCConnectionOptions` object with these properties:

- `Vendor` — Database vendor name
- `DataSourceName` — Name of the data source
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

- `AuthType` — Authentication type
- `JDBCDriverLocation` — Full path of the JDBC driver file

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, database server `dbtb04`, port number `54317`, full path to the JDBC driver file, and Windows® authentication.

```
opts = setConnectionOptions(opts, ...
    'DataSourceName', 'SQLServerDataSource', ...
    'Server', 'dbtb04', 'PortNumber', 54317, ...
    'JDBCDriverLocation', 'C:\Drivers\sqljdbc4.jar', ...
    'AuthType', 'Windows')
```

```
opts =
  JDBCConnectionOptions with properties:

      Vendor: 'Microsoft SQL Server'
DataSourceName: 'SQLServerDataSource'

      DatabaseName: ''
      Server: 'dbtb04'
      PortNumber: 54317
      AuthType: 'Windows'

      JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

The `setConnectionOptions` function sets the `DataSourceName`, `Server`, `PortNumber`, `AuthType`, and `JDBCDriverLocation` properties in the `JDBCConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)
```

```
status = logical
```

```
1
```

Save the configured data source.

```
saveAsJDBCDataSource(opts)
```

Delete the configured data source.

```
datasource = opts.DataSourceName;
deleteJDBCDataSource(datasource)
```

Input Arguments

datasource — Data source name

character vector | string scalar

Data source name, specified as a character vector or string scalar. Specify the name of an existing configured JDBC data source.

Example: "myJDBCDataSource"

Data Types: char | string

Alternative Functionality

App

You can delete a JDBC data source by using the JDBC Data Source Configuration dialog box in the **Database Explorer** app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure JDBC data source**.

Version History

Introduced in R2019b

See Also

Objects

JDBCConnectionOptions

Functions

configureJDBCDataSource | saveAsJDBCDataSource | deleteDataSource

Topics

"Modify and Delete Data Sources" on page 4-17

"Configure Driver and Data Source" on page 2-14

rmConnectionOptions

Package: database.options

(To be removed) Remove JDBC driver-specific connection options

Note The `rmConnectionOptions` function will be removed in a future release. Use the `rmoptions` function instead. For details, see “Compatibility Considerations”.

Syntax

```
opts = rmConnectionOptions(opts,option)
```

Description

`opts = rmConnectionOptions(opts,option)` removes a JDBC driver-specific connection option using the `JDBCConnectionOptions` object `opts`.

Examples

Remove Driver-Specific Connection Option

Create a JDBC data source for a Microsoft SQL Server database, configure the data source by setting JDBC connection options, and set and remove an additional JDBC driver-specific option. Then, test and save the data source.

Create an SQL Server data source.

```
opts = configureJDBCDataSource('Vendor','Microsoft SQL Server')
```

```
opts =
```

```
  JDBCConnectionOptions with properties:
```

```
      Vendor: 'Microsoft SQL Server'
DataSourceName: ''
      DatabaseName: ''
      Server: 'localhost'
      PortNumber: 1433
      AuthType: 'Server'
```

```
  JDBCDriverLocation: ''
```

`opts` is a `JDBCConnectionOptions` object with these properties:

- `Vendor` — Database vendor name
- `DataSourceName` — Name of the data source

- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number
- `AuthType` — Authentication type
- `JDBCDriverLocation` — Full path of the JDBC driver file

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, database server `dbtb04`, port number `54317`, full path to the JDBC driver file, and Windows® authentication.

```
opts = setConnectionOptions(opts, ...
    'DataSourceName','SQLServerDataSource', ...
    'Server','dbtb04','PortNumber',54317, ...
    'JDBCDriverLocation','C:\Drivers\sqljdbc4.jar', ...
    'AuthType','Windows')
```

```
opts =
```

```
JDBCConnectionOptions with properties:
```

```
Vendor: 'Microsoft SQL Server'
DataSourceName: 'SQLServerDataSource'
```

```
DatabaseName: ''
Server: 'dbtb04'
PortNumber: 54317
AuthType: 'Windows'
```

```
JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

The `setConnectionOptions` function sets the `DataSourceName`, `Server`, `PortNumber`, `AuthType`, and `JDBCDriverLocation` properties in the `JDBCConnectionOptions` object.

Add a JDBC driver-specific connection option by using a name-value pair argument. The option specifies a timeout value for establishing the database connection. `opts` contains a new section of properties for the additional JDBC connection option.

```
opts = addConnectionOptions(opts, 'loginTimeout',20)
```

```
opts =
```

```
JDBCConnectionOptions with properties:
```

```
Vendor: 'Microsoft SQL Server'
DataSourceName: 'SQLServerDataSource'
```

```
DatabaseName: ''
Server: 'dbtb04'
PortNumber: 54317
AuthType: 'Windows'
```

```
JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

```
Additional JDBC Connection Options:
```

```
loginTimeout: '20'
```

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates that the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts,username,password)

status = logical

    1
```

Remove the JDBC driver-specific option. The `opts` object no longer contains the properties section for the additional JDBC connection options.

```
opts = rmConnectionOptions(opts,'loginTimeout')

opts =

    JDBCConnectionOptions with properties:

        Vendor: 'Microsoft SQL Server'
    DataSourceName: 'SQLServerDataSource'

        DatabaseName: ''
        Server: 'dbtb04'
        PortNumber: 54317
        AuthType: 'Windows'

    JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

Test the database connection again.

```
status = testConnection(opts,username,password)

status = logical

    1
```

Save the configured data source.

```
saveAsJDBCDataSource(opts)
```

Input Arguments

opts — JDBC connection options

JDBCConnectionOptions object

JDBC connection options, specified as a JDBCConnectionOptions object.

option — JDBC driver-specific option

character vector | string scalar | cell array of character vectors | string array

JDBC driver-specific option, specified as a character vector, string scalar, cell array of character vectors, or string array. Specify the name of one or more JDBC driver-specific connection options that you added using the `addConnectionOptions` function.

Example: `"loginTimeout"`

Data Types: `char | string | cell`

Output Arguments

opts — JDBC connection options

JDBCConnectionOptions object

JDBC connection options, returned as a JDBCConnectionOptions object.

Alternative Functionality

App

You can remove JDBC driver-specific connection options by using the JDBC Data Source Configuration dialog box in the **Database Explorer** app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure JDBC data source**.

Version History

Introduced in R2019b

R2020b: rmConnectionOptions function will be removed

Not recommended starting in R2020b

The `rmConnectionOptions` function will be removed in a future release. Use the `rmoptions` function instead. Some differences between the workflows might require updates to your code.

Update Code

Use the `rmoptions` function with the `SQLConnectionOptions` object to remove JDBC driver-specific connection options.

In prior releases, you configured a JDBC data source using the `JDBCConnectionOptions` object, and removed options using the `rmConnectionOptions` function. For example:

```
opts = configureJDBCDataSource('Vendor','Microsoft SQL Server');
opts = setConnectionOptions(opts, ...
    'DataSourceName','SQLServerDataSource', ...
    'Server','dbtb04','PortNumber',54317, ...
    'JDBCDriverLocation','C:\Drivers\sqljdbc4.jar', ...
    'AuthType','Windows');
opts = addConnectionOptions(opts,'loginTimeout',20);
username = "";
password = "";
status = testConnection(opts,username,password);
opts = rmConnectionOptions(opts,'loginTimeout');
status = testConnection(opts,username,password);
saveAsJDBCDataSource(opts)
```

Now you can set JDBC driver-specific connection options with the `SQLConnectionOptions` object instead, and then remove options using the `rmoptions` function .

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc",vendor);
opts = setoptions(opts, ...
    'DataSourceName','SQLServerDataSource', ...
    'JDBCDriverLocation','C:\Drivers\sqljdbc4.jar', ...
    'DatabaseName','toystore_doc','Server','dbtb04', ...
    'PortNumber',54317,'AuthType','Windows', ...
    'loginTimeout',20);
username = "";
password = "";
status = testConnection(opts,username,password);
opts = rmoptions(opts,'loginTimeout');
status = testConnection(opts,username,password);
saveAsDataSource(opts)
```

See Also

Objects

JDBCConnectionOptions

Functions

[configureJDBCDataSource](#) | [saveAsJDBCDataSource](#) | [addConnectionOptions](#) | [setConnectionOptions](#) | [testConnection](#) | [deleteJDBCDataSource](#) | [rmoptions](#)

Topics

[“Modify and Delete Data Sources”](#) on page 4-17

[“Configure Driver and Data Source”](#) on page 2-14

saveAsJDBCDataSource

Package: database.options

(To be removed) Save JDBC data source

Note The `saveAsJDBCDataSource` function will be removed in a future release. Use the `saveAsDataSource` function instead. For details, see “Compatibility Considerations”.

Syntax

```
saveAsJDBCDataSource(opts)
```

Description

`saveAsJDBCDataSource(opts)` saves the JDBC data source specified by the `JDBCConnectionOptions` object `opts`.

Examples

Create JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft SQL Server database.

Create an SQL Server data source.

```
opts = configureJDBCDataSource('Vendor', 'Microsoft SQL Server')
```

```
opts =
```

```
  JDBCConnectionOptions with properties:
```

```
      Vendor: 'Microsoft SQL Server'
DataSourceName: ''
      DatabaseName: ''
      Server: 'localhost'
      PortNumber: 1433
      AuthType: 'Server'
```

```
      JDBCDriverLocation: ''
```

`opts` is a `JDBCConnectionOptions` object with these properties:

- `Vendor` — Database vendor name
- `DataSourceName` — Name of the data source
- `DatabaseName` — Name of the database
- `Server` — Name of the database server

- `PortNumber` — Port number
- `AuthType` — Authentication type
- `JDBCDriverLocation` — Full path of the JDBC driver file

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, database server `dbtb04`, port number `54317`, full path to the JDBC driver file, and Windows® authentication.

```
opts = setConnectionOptions(opts, ...
    'DataSourceName', 'SQLServerDataSource', ...
    'Server', 'dbtb04', 'PortNumber', 54317, ...
    'JDBCDriverLocation', 'C:\Drivers\sqljdbc4.jar', ...
    'AuthType', 'Windows')
```

```
opts =
```

```
JDBCConnectionOptions with properties:
```

```
Vendor: 'Microsoft SQL Server'
DataSourceName: 'SQLServerDataSource'
```

```
DatabaseName: ''
Server: 'dbtb04'
PortNumber: 54317
AuthType: 'Windows'
```

```
JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

The `setConnectionOptions` function sets the `DataSourceName`, `Server`, `PortNumber`, `AuthType`, and `JDBCDriverLocation` properties in the `JDBCConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)
```

```
status = logical
```

```
1
```

Save the configured data source.

```
saveAsJDBCDataSource(opts)
```

You can connect to the new data source using the `database` function or the Database Explorer app.

Input Arguments

opts — JDBC connection options

`JDBCConnectionOptions` object

JDBC connection options, specified as a `JDBCConnectionOptions` object.

Alternative Functionality

App

You can save a JDBC data source by using the JDBC Data Source Configuration dialog box in the **Database Explorer** app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure JDBC data source**.

Version History

Introduced in R2019b

R2020b: saveAsJDBCDataSource function will be removed

Not recommended starting in R2020b

The saveAsJDBCDataSource function will be removed in a future release. Use the saveAsDataSource function instead. Some differences between the workflows might require updates to your code.

Update Code

Use the saveAsDataSource function with the SqlConnectionOptions object to save the JDBC data source.

In prior releases, you configured and saved a JDBC data source using the JDBCConnectionOptions object and the saveAsJDBCDataSource function. For example:

```
opts = configureJDBCDataSource('Vendor','Microsoft SQL Server');
opts = setConnectionOptions(opts, ...
    'DataSourceName','SQLServerDataSource', ...
    'Server','dbtb04','PortNumber',54317, ...
    'JDBCDriverLocation','C:\Drivers\sqljdbc4.jar', ...
    'AuthType','Windows');
username = "";
password = "";
status = testConnection(opts,username,password);
saveAsJDBCDataSource(opts)
```

Now you can set JDBC connection options and save the data source using the SqlConnectionOptions object and the saveAsDataSource function instead.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc",vendor);
opts = setoptions(opts, ...
    'DataSourceName',"SQLServerDataSource", ...
    'JDBCDriverLocation',"C:\Drivers\sqljdbc4.jar", ...
    'DatabaseName',"toystore_doc",'Server',"dbtb04", ...
    'PortNumber',54317,'AuthType',"Windows");
username = "";
password = "";
status = testConnection(opts,username,password);
saveAsDataSource(opts)
```

See Also

Objects

JDBCConnectionOptions

Functions

configureJDBCDataSource | addConnectionOptions | setConnectionOptions |
rmConnectionOptions | testConnection | deleteJDBCDataSource | saveAsDataSource

Topics

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

setConnectionOptions

Package: database.options

(To be removed) Set JDBC connection options

Note The `setConnectionOptions` function will be removed in a future release. Use the `setOptions` function instead. For details, see “Compatibility Considerations”.

Syntax

```
opts = setConnectionOptions(opts,
Option1,OptionValue1,...,OptionN,OptionValueN)
```

Description

`opts = setConnectionOptions(opts, Option1,OptionValue1,...,OptionN,OptionValueN)` sets JDBC connection options using the `JDBCConnectionOptions` object `opts`.

Examples

Create JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft SQL Server database.

Create an SQL Server data source.

```
opts = configureJDBCDataSource('Vendor', 'Microsoft SQL Server')
```

```
opts =
```

```
  JDBCConnectionOptions with properties:
```

```
      Vendor: 'Microsoft SQL Server'
DataSourceName: ''
```

```
  DatabaseName: ''
      Server: 'localhost'
  PortNumber: 1433
      AuthType: 'Server'
```

```
  JDBCDriverLocation: ''
```

`opts` is a `JDBCConnectionOptions` object with these properties:

- `Vendor` — Database vendor name
- `DataSourceName` — Name of the data source
- `DatabaseName` — Name of the database

- `Server` — Name of the database server
- `PortNumber` — Port number
- `AuthType` — Authentication type
- `JDBCDriverLocation` — Full path of the JDBC driver file

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, database server `dbtb04`, port number `54317`, full path to the JDBC driver file, and `Windows®` authentication.

```
opts = setConnectionOptions(opts, ...
    'DataSourceName', 'SQLServerDataSource', ...
    'Server', 'dbtb04', 'PortNumber', 54317, ...
    'JDBCDriverLocation', 'C:\Drivers\sqljdbc4.jar', ...
    'AuthType', 'Windows')
```

```
opts =
```

```
JDBCConnectionOptions with properties:
```

```
Vendor: 'Microsoft SQL Server'
DataSourceName: 'SQLServerDataSource'
```

```
DatabaseName: ''
Server: 'dbtb04'
PortNumber: 54317
AuthType: 'Windows'
```

```
JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

The `setConnectionOptions` function sets the `DataSourceName`, `Server`, `PortNumber`, `AuthType`, and `JDBCDriverLocation` properties in the `JDBCConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts,username,password)
```

```
status = logical
```

```
1
```

Save the configured data source.

```
saveAsJDBCDataSource(opts)
```

You can connect to the new data source using the `database` function or the Database Explorer app.

Input Arguments

`opts` — JDBC connection options

`JDBCConnectionOptions` object

JDBC connection options, specified as a `JDBCConnectionOptions` object.

Option1,OptionValue1,...,OptionN,OptionValueN — JDBC connection options to set name-value pair arguments

JDBC connection options to set, specified as one or more name-value pair arguments. `Option` is a character vector or string scalar that specifies the name of a JDBC connection option. `OptionValue` specifies the value of the JDBC connection option. `OptionValue` can be a character vector, string scalar, logical scalar, or numeric scalar. You can specify any JDBC connection option that is a property of the `JDBCConnectionOptions` object.

Example:

```
'DataSourceName','myDataSource','Vendor','MySQL','Server','localhost','PortNumber',3306
```

configures a JDBC data source named `myDataSource` for a MySQL database located on the local server with the port number 3306.

Output Arguments

opts — JDBC connection options

`JDBCConnectionOptions` object

JDBC connection options, returned as a `JDBCConnectionOptions` object.

Alternative Functionality

App

You can set JDBC connection options by using the JDBC Data Source Configuration dialog box in the **Database Explorer** app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure JDBC data source**.

Version History

Introduced in R2019b

R2020b: setConnectionOptions function will be removed

Not recommended starting in R2020b

The `setConnectionOptions` function will be removed in a future release. Use the `setoptions` function instead. Some differences between the workflows might require updates to your code.

Update Code

Use the `setoptions` function with the `SQLConnectionOptions` object to set the JDBC connection options.

In prior releases, you configured a JDBC data source using the `setConnectionOptions` function and the `JDBCConnectionOptions` object. For example:

```
opts = configureJDBCDataSource('Vendor','Microsoft SQL Server');
opts = setConnectionOptions(opts, ...
    'DataSourceName','SQLServerDataSource', ...
    'Server','dbtb04','PortNumber',54317, ...
    'JDBCDriverLocation','C:\Drivers\sqljdbc4.jar', ...
    'AuthType','Windows');
username = "";
```

```
password = "";  
status = testConnection(opts,username,password);  
saveAsJDBCDataSource(opts)
```

Now you can set JDBC connection options using the `setoptions` function, and save the data source using the `SQLConnectionOptions` object instead.

```
vendor = "Microsoft SQL Server";  
opts = databaseConnectionOptions("jdbc",vendor);  
opts = setoptions(opts, ...  
    'DataSourceName','SQLServerDataSource', ...  
    'JDBCDriverLocation','C:\Drivers\sqljdbc4.jar', ...  
    'DatabaseName','toystore_doc','Server','dbtb04', ...  
    'PortNumber',54317,'AuthType','Windows');  
username = "";  
password = "";  
status = testConnection(opts,username,password);  
saveAsDataSource(opts)
```

See Also

Objects

JDBCConnectionOptions

Functions

`configureJDBCDataSource` | `saveAsJDBCDataSource` | `addConnectionOptions` | `rmConnectionOptions` | `testConnection` | `deleteJDBCDataSource` | `setoptions`

Topics

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

testConnection

Package: database.options

(To be removed) Test JDBC data source connection

Note The `testConnection` function will be removed in a future release. Use the `testConnection` function of the `SQLConnectionOptions` object instead. For details, see “Compatibility Considerations”.

Syntax

```
status = testConnection(opts,username,password)
[status,message] = testConnection(opts,username,password)
```

Description

`status = testConnection(opts,username,password)` tests the JDBC data source connection specified by the `JDBCConnectionOptions` object `opts`, a user name, and a password.

`[status,message] = testConnection(opts,username,password)` also returns the error message associated with testing the database connection.

Examples

Create JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft SQL Server database.

Create an SQL Server data source.

```
opts = configureJDBCDataSource('Vendor','Microsoft SQL Server')
```

```
opts =
```

```
    JDBCConnectionOptions with properties:
```

```
        Vendor: 'Microsoft SQL Server'
DataSourceName: ''
```

```
        DatabaseName: ''
           Server: 'localhost'
        PortNumber: 1433
           AuthType: 'Server'
```

```
        JDBCDriverLocation: ''
```

`opts` is a `JDBCConnectionOptions` object with these properties:

- `Vendor` — Database vendor name

- `DataSourceName` — Name of the data source
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number
- `AuthType` — Authentication type
- `JDBCDriverLocation` — Full path of the JDBC driver file

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, database server `dbtb04`, port number `54317`, full path to the JDBC driver file, and Windows® authentication.

```
opts = setConnectionOptions(opts, ...
    'DataSourceName', 'SQLServerDataSource', ...
    'Server', 'dbtb04', 'PortNumber', 54317, ...
    'JDBCDriverLocation', 'C:\Drivers\sqljdbc4.jar', ...
    'AuthType', 'Windows')
```

```
opts =
```

```
JDBCConnectionOptions with properties:
```

```
Vendor: 'Microsoft SQL Server'
DataSourceName: 'SQLServerDataSource'
```

```
DatabaseName: ''
Server: 'dbtb04'
PortNumber: 54317
AuthType: 'Windows'
```

```
JDBCDriverLocation: 'C:\Drivers\sqljdbc4.jar'
```

The `setConnectionOptions` function sets the `DataSourceName`, `Server`, `PortNumber`, `AuthType`, and `JDBCDriverLocation` properties in the `JDBCConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)
```

```
status = logical
```

```
1
```

Save the configured data source.

```
saveAsJDBCDataSource(opts)
```

You can connect to the new data source using the `database` function or the Database Explorer app.

Input Arguments

opts — JDBC connection options

JDBCConnectionOptions object

JDBC connection options, specified as a JDBCConnectionOptions object.

username — User name

character vector | string scalar

User name required to access the database, specified as a character vector or string scalar. If no user name is required, specify an empty value `''`.

Data Types: `char` | `string`

password — Password

character vector | string scalar

Password required to access the database, specified as a character vector or string scalar. If no password is required, specify an empty value `''`.

Data Types: `char` | `string`

Output Arguments

status — Connection status

logical

Connection status, returned as a logical `true` if the connection test passes or a logical `false` if the connection test fails.

message — Error message

character vector

Error message, returned as a character vector. If the connection test passes, then the error message is an empty character vector. Otherwise, the error message contains text from the resulting failure in database connection.

Alternative Functionality

App

You can test a JDBC data source connection by using the JDBC Data Source Configuration dialog box in the **Database Explorer** app. In the **Data Source** section of the **Database Explorer** tab, select **Configure Data Source > Configure JDBC data source**.

Version History

Introduced in R2019b

R2020b: testConnection function will be removed

Not recommended starting in R2020b

The `testConnection` function will be removed in a future release. Use the `testConnection` function of the `SQLConnectionOptions` object instead. Some differences between the workflows might require updates to your code.

Update Code

Use the `testConnection` function with the `SQLConnectionOptions` object to test the specified JDBC connection options.

In prior releases, you configured a JDBC data source using the `JDBCConnectionOptions` object. For example:

```
opts = configureJDBCDataSource('Vendor', 'Microsoft SQL Server');
opts = setConnectionOptions(opts, ...
    'DataSourceName', 'SQLServerDataSource', ...
    'Server', 'dbtb04', 'PortNumber', 54317, ...
    'JDBCDriverLocation', 'C:\Drivers\sqljdbc4.jar', ...
    'AuthType', 'Windows');
username = "";
password = "";
status = testConnection(opts, username, password);
saveAsJDBCDataSource(opts)
```

Now you can set JDBC connection options and save the data source using the `SQLConnectionOptions` object instead.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc", vendor);
opts = setoptions(opts, ...
    'DataSourceName', "SQLServerDataSource", ...
    'JDBCDriverLocation', "C:\Drivers\sqljdbc4.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...
    'PortNumber', 54317, 'AuthType', "Windows");
username = "";
password = "";
status = testConnection(opts, username, password);
saveAsDataSource(opts)
```

See Also

Objects

`JDBCConnectionOptions`

Functions

`configureJDBCDataSource` | `saveAsJDBCDataSource` | `addConnectionOptions` | `setConnectionOptions` | `rmConnectionOptions` | `deleteJDBCDataSource` | `testConnection`

Topics

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

SQLConnectionOptions

Define JDBC database connection options

Description

Create connection options for a JDBC database connection.

After you create an `SQLConnectionOptions` object, set the connection options, test the connection, and save the data source, you can create a JDBC database connection using the saved data source. The connection options include the options required to make a database connection. You can also define additional connection options for a specific database driver.

Creation

Create an `SQLConnectionOptions` object using the `databaseConnectionOptions` function.

Properties

All Databases

DataSourceName — Data source name

string scalar

Data source name, specified as a string scalar. You can use the data source name in the `database` function to create a JDBC database connection.

Example: "MSSQLServer"

Data Types: string

Vendor — Database vendor

string scalar

This property is read-only.

Database vendor, specified as a string scalar. Set this property using the `vendor` input argument in the `databaseConnectionOptions` function.

Example: "Microsoft SQL Server"

Data Types: string

JDBCDriverLocation — JDBC driver location

string scalar

JDBC driver location, specified as a string scalar. Specify the full path to the JDBC driver file, including the name of the file.

Example: "C:\drivers\sqljdbc4.jar"

Data Types: string

Common Properties for Microsoft SQL Server, MySQL, Oracle, and PostgreSQL Databases

DatabaseName — Database name

string scalar

Database name on the server, specified as a string scalar.

Example: "mydatabase"

Data Types: string

Server — Database server name or address

"localhost" (default) | string scalar

Database server name or address, specified as a string scalar.

Data Types: string

PortNumber — Server port number where the server is listening

numeric scalar

Server port number where the server is listening, specified as a numeric scalar. The default value is based on the database vendor:

- Microsoft SQL Server — 1433
- MySQL — 3306
- Oracle — 1521
- PostgreSQL — 5432

Data Types: double

Microsoft SQL Server Database Only

AuthenticationType — Authentication type

"Server" (default) | "Windows"

Authentication type, specified as one of these values:

- "Server" — Microsoft SQL Server authentication
- "Windows" — Windows authentication

Specify the value as a string scalar.

Oracle Database Only

DriverType — Driver type

"thin" (default) | "oci"

Driver type, specified as one of these values:

- "thin" — Thin driver
- "oci" — Windows authentication or OCI driver

Specify the value as a string scalar.

Other Databases

Driver — JDBC driver name

string scalar

JDBC driver name, specified as a string scalar that refers to the Java driver that implements the `java.sql.Driver` interface.

For details about the JDBC driver name, consult your database driver documentation.

Example: `org.sqlite.JDBC`

Data Types: `string`

URL — Database connection URL

string scalar

Database connection URL, specified as a string scalar for the vendor-specific URL. This URL is typically constructed using connection properties such as the server name, port number, and database name.

For details about the database connection URL, consult your database driver documentation.

Example: `jdbc:sqlite:C:\Databases\sqlite.db`

Data Types: `string`

Object Functions

<code>setoptions</code>	Set JDBC connection options
<code>rmoptions</code>	Remove JDBC connection options
<code>reset</code>	Reset JDBC connection options to defaults
<code>testConnection</code>	Test JDBC database connection
<code>saveAsDataSource</code>	Save JDBC data source

Examples

Create JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft® SQL Server® database.

Create an SQL Server data source for a JDBC database connection.

```
vendor = "Microsoft SQL Server";  
opts = databaseConnectionOptions("jdbc", vendor)
```

```
opts =  
    SqlConnectionOptions with properties:
```

```
        DataSourceName: ""  
        Vendor: "Microsoft SQL Server"  
  
        JDBCdriverLocation: ""  
        DatabaseName: ""  
        Server: "localhost"  
        PortNumber: 1433
```

```
AuthenticationType: "Server"
```

opts is an SQLConnectionOptions object with these properties:

- DataSourceName — Name of the data source
- Vendor — Database vendor name
- JDBCDriverLocation — Full path of the JDBC driver file
- DatabaseName — Name of the database
- Server — Name of the database server
- PortNumber — Port number
- AuthenticationType — Authentication type

Configure the data source by setting the JDBC connection options for the data source SQLServerDataSource, full path to the JDBC driver file, database name toystore_doc, database server dbtb04, port number 54317, and Windows® authentication.

```
opts = setoptions(opts, ...
    'DataSourceName', "SQLServerDataSource", ...
    'JDBCDriverLocation', "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...
    'PortNumber', 54317, 'AuthenticationType', "Windows")
```

```
opts =
    SQLConnectionOptions with properties:
```

```
        DataSourceName: "SQLServerDataSource"
        Vendor: "Microsoft SQL Server"

        JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
        DatabaseName: "toystore_doc"
        Server: "dbtb04"
        PortNumber: 54317
        AuthenticationType: "Windows"
```

The setoptions function sets the DataSourceName, JDBCDriverLocation, DatabaseName, Server, PortNumber, and AuthenticationType properties in the SQLConnectionOptions object.

Test the database connection with a blank user name and password. The testConnection function returns the logical 1, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)

status = logical
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the database function or the Database Explorer app.

Version History

Introduced in R2020b

See Also

databaseConnectionOptions | deleteDataSource

Topics

“Create JDBC Data Source and Set Options Programmatically” on page 2-17

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

databaseConnectionOptions

Create database connection options

Syntax

```
opts = databaseConnectionOptions(drivertype,vendor)
opts = databaseConnectionOptions(datasource)
```

Description

`opts = databaseConnectionOptions(drivertype,vendor)` creates an `SQLConnectionOptions` object `opts` using the specified driver type and database vendor. The `SQLConnectionOptions` object contains the database connection options.

`opts = databaseConnectionOptions(datasource)` enables you to edit an existing data source using its name.

Examples

Create JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft® SQL Server® database.

Create an SQL Server data source for a JDBC database connection.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc",vendor)
```

```
opts =
  SQLConnectionOptions with properties:
      DataSourceName: ""
      Vendor: "Microsoft SQL Server"
      JDBCDriverLocation: ""
      DatabaseName: ""
      Server: "localhost"
      PortNumber: 1433
      AuthenticationType: "Server"
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `JDBCDriverLocation` — Full path of the JDBC driver file
- `DatabaseName` — Name of the database
- `Server` — Name of the database server

- `PortNumber` — Port number
- `AuthenticationType` — Authentication type

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, full path to the JDBC driver file, database name `toystore_doc`, database server `dbtb04`, port number `54317`, and Windows® authentication.

```
opts = setoptions(opts, ...
    'DataSourceName', "SQLServerDataSource", ...
    'JDBCDriverLocation', "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...
    'PortNumber', 54317, 'AuthenticationType', "Windows")
```

```
opts =
  SqlConnectionOptions with properties:

      DataSourceName: "SQLServerDataSource"
      Vendor: "Microsoft SQL Server"

      JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
      DatabaseName: "toystore_doc"
      Server: "dbtb04"
      PortNumber: 54317
      AuthenticationType: "Windows"
```

The `setoptions` function sets the `DataSourceName`, `JDBCDriverLocation`, `DatabaseName`, `Server`, `PortNumber`, and `AuthenticationType` properties in the `SqlConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)

status = logical
      1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `database` function or the Database Explorer app.

Edit Existing JDBC Data Source

Edit an existing JDBC data source for a Microsoft® SQL Server® database. Set an additional JDBC driver-specific option, and save the data source.

Retrieve the existing SQL Server data source `SQLServerDataSource`.

```

datasource = "SQLServerDataSource";
opts = databaseConnectionOptions(datasource)

opts =
  SqlConnectionOptions with properties:

      DataSourceName: "SQLServerDataSource"
      Vendor: "Microsoft SQL Server"

      JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
      DatabaseName: "toystore_doc"
      Server: "dbtb04"
      PortNumber: 54317
      AuthenticationType: "Windows"

```

opts is an SqlConnectionOptions object with these properties:

- DataSourceName — Name of the data source
- Vendor — Database vendor name
- JDBCDriverLocation — Full path of the JDBC driver file
- DatabaseName — Name of the database
- Server — Name of the database server
- PortNumber — Port number
- AuthenticationType — Authentication type

Add a JDBC driver-specific connection option by using a name-value pair argument. The option specifies a timeout value for establishing the database connection. opts contains a new section of properties for the additional JDBC connection option.

```

opts = setoptions(opts, 'loginTimeout', "20")

opts =
  SqlConnectionOptions with properties:

      DataSourceName: "SQLServerDataSource"
      Vendor: "Microsoft SQL Server"

      JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
      DatabaseName: "toystore_doc"
      Server: "dbtb04"
      PortNumber: 54317
      AuthenticationType: "Windows"

  Additional Connection Options:

      loginTimeout: "20"

```

Test the database connection with a blank user name and password. The testConnection function returns the logical 1, which indicates the database connection is successful.

```

username = "";
password = "";
status = testConnection(opts,username,password)

```

```
status = logical  
      1
```

Save the updated data source.

```
saveAsDataSource(opts)
```

Input Arguments

drvertype — Driver type

```
"jdbc" | "native"
```

Driver type, specified as one of these values:

- "jdbc" — JDBC driver
- "native" — Native interface

You can specify the value as either a character vector or string scalar.

vendor — Database vendor

```
"Microsoft SQL Server" | "MySQL" | "Oracle" | "PostgreSQL" | "Other" | ...
```

Database vendor, specified as one of these values:

- For JDBC drivers:
 - "Microsoft SQL Server" — Microsoft SQL Server database
 - "MySQL" — MySQL database
 - "Oracle" — Oracle database
 - "PostgreSQL" — PostgreSQL database
 - "Other" — Other database
- For native interfaces:
 - "MySQL" — MySQL native interface
 - "PostgreSQL" — PostgreSQL native interface
 - "Cassandra" — Apache Cassandra database C++ interface

If the `drvertype` input argument is "jdbc", then the `vendor` argument must be one of the values for JDBC drivers. If the `drvertype` input argument is "native", then the `vendor` argument must be one of the values for native interfaces.

You can specify the value as either a character vector or a string scalar.

datasource — Data source name

```
character vector | string scalar
```

Data source name, specified as a character vector or string scalar. Specify the name of an existing data source.

Example: "myDataSource"

Data Types: char | string

Output Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, returned as one of the following objects:

- SQLConnectionOptions — JDBC database connection options
- SQLConnectionOptions — MySQL native interface connection options
- SQLConnectionOptions — PostgreSQL native interface connection options
- CassandraConnectionOptions — Apache Cassandra database connection options

Alternative Functionality

App

You can open the JDBC Data Source Configuration, MySQL Data Source Configuration, or PostgreSQL Data Source Configuration dialog boxes using the **Database Explorer** app. In the **Data Source** section of the **Database Explorer** tab, select one of these accordingly:

- **Configure Data Source > Configure JDBC data source**
- **Configure Data Source > Configure native data source > MySQL**
- **Configure Data Source > Configure native data source > PostgreSQL**

Version History

Introduced in R2020b

See Also

Apps

Database Explorer

Objects

SQLConnectionOptions

Functions

listDataSources | setoptions | saveAsDataSource | testConnection | database | deleteDataSource

Topics

“Create JDBC Data Source and Set Options Programmatically” on page 2-17

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

deleteDataSource

Delete data source

Syntax

```
deleteDataSource(datasource)
```

Description

deleteDataSource(datasource) deletes the specified data source.

Examples

Delete Existing JDBC Data Source

Delete an existing JDBC data source for a Microsoft® SQL Server® database.

List the configured data sources on the system. The JDBC data source `SQLServerDataSource` is an existing data source on the system.

```
listDataSources
```

ans=3×3 table

Name	DriverType	Vendor
"MSSQLServerAuth"	"ODBC"	<missing>
"MSSQLServerJDBCAuth"	"JDBC"	"Microsoft SQL Server"
"SQLServerDataSource"	"JDBC"	"Microsoft SQL Server"

Delete the SQL Server data source `SQLServerDataSource`.

```
datasource = "SQLServerDataSource";
deleteDataSource(datasource)
```

List the configured data sources again. The list of existing data sources does not contain the `SQLServerDataSource` data source.

```
listDataSources
```

ans=2×3 table

Name	DriverType	Vendor
"MSSQLServerAuth"	"ODBC"	<missing>
"MSSQLServerJDBCAuth"	"JDBC"	"Microsoft SQL Server"

Input Arguments

datasource — Data source name

character vector | string scalar

Data source name, specified as a character vector or string scalar. Specify the name of an existing data source.

Example: "myDataSource"

Data Types: char | string

Version History

Introduced in R2020b

See Also

Objects

SQLConnectionOptions

Functions

databaseConnectionOptions | setoptions | roptions | reset | saveAsDataSource | database | testConnection | listDataSources

Topics

"Create JDBC Data Source and Set Options Programmatically" on page 2-17

"Modify and Delete Data Sources" on page 4-17

"Configure Driver and Data Source" on page 2-14

reset

Package: `database.options.jdbc.sqlserver`

Reset JDBC connection options to defaults

Syntax

```
opts = reset(opts)
```

Description

`opts = reset(opts)` resets all JDBC connection options to their default values for all properties of the `SQLConnectionOptions` object. The `reset` function also removes any additional JDBC driver-specific properties from the `SQLConnectionOptions` object.

Examples

Reset Connection Options to Default Values

Create, configure, and test a JDBC data source for a Microsoft® SQL Server® database. Reset the JDBC connection options to their default values. Then configure, test, and save the data source with different JDBC connection options.

Create an SQL Server data source for a JDBC database connection.

```
vendor = "Microsoft SQL Server";  
opts = databaseConnectionOptions("jdbc", vendor)
```

```
opts =  
  SQLConnectionOptions with properties:  
    DataSourceName: ""  
    Vendor: "Microsoft SQL Server"  
  
    JDBCDriverLocation: ""  
    DatabaseName: ""  
    Server: "localhost"  
    PortNumber: 1433  
    AuthenticationType: "Server"
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `JDBCDriverLocation` — Full path of the JDBC driver file
- `DatabaseName` — Name of the database
- `Server` — Name of the database server

- PortNumber — Port number
- AuthenticationType — Authentication type

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, full path to the JDBC driver file, database name `toystore_doc`, database server `dbtb04`, port number `54317`, and Windows® authentication.

```
opts = setoptions(opts, ...
    'DataSourceName', "SQLServerDataSource", ...
    'JDBCDriverLocation', "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...
    'PortNumber', 54317, 'AuthenticationType', "Windows")
```

```
opts =
  SqlConnectionOptions with properties:

      DataSourceName: "SQLServerDataSource"
      Vendor: "Microsoft SQL Server"

      JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
      DatabaseName: "toystore_doc"
      Server: "dbtb04"
      PortNumber: 54317
      AuthenticationType: "Windows"
```

The `setoptions` function sets the `DataSourceName`, `JDBCDriverLocation`, `DatabaseName`, `Server`, `PortNumber`, and `AuthenticationType` properties in the `SqlConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)

status = logical
      1
```

Reset the JDBC connection options to their default values. The properties of the `SqlConnectionOptions` object contain the default values.

```
opts = reset(opts)

opts =
  SqlConnectionOptions with properties:

      DataSourceName: ""
      Vendor: "Microsoft SQL Server"

      JDBCDriverLocation: ""
      DatabaseName: ""
      Server: "localhost"
      PortNumber: 1433
      AuthenticationType: "Server"
```

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, full path to the JDBC driver file, database name `toystore_doc`, database server `dbtb04`, port number `54317`, and Windows® authentication. Also, set driver-specific connection options to specify a timeout value for establishing the database connection, and to disable SSL encryption.

```
opts = setoptions(opts, ...
    "DataSourceName", "SQLServerDataSource", ...
    "JDBCDriverLocation", ...
    "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar", ...
    "DatabaseName", "toystore_doc", ...
    "Server", "dbtb04", "PortNumber", 54317, ...
    "AuthenticationType", "Windows", "loginTimeout", "20", ...
    "encrypt", "false")

opts =
    SqlConnectionOptions with properties:

        DataSourceName: "SQLServerDataSource"
        Vendor: "Microsoft SQL Server"

        JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
        DatabaseName: "toystore_doc"
        Server: "dbtb04"
        PortNumber: 54317
        AuthenticationType: "Windows"

    Additional Connection Options:

        encrypt: "false"
        loginTimeout: "20"
```

The `setoptions` function sets the `DataSourceName`, `JDBCDriverLocation`, `DatabaseName`, `Server`, `PortNumber`, and `AuthenticationType` properties in the `SqlConnectionOptions` object. The driver-specific connection options appear below the other connection options.

Test the database connection again.

```
username = "";
password = "";
status = testConnection(opts, username, password)

status = logical
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

`SqlConnectionOptions` object

Database connection options, specified as an `SqlConnectionOptions` object.

Output Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, returned as an SQLConnectionOptions object.

Version History

Introduced in R2020b

See Also

Objects

SQLConnectionOptions

Functions

databaseConnectionOptions | setoptions | roptions | saveAsDataSource | database | deleteDataSource | testConnection

Topics

“Create JDBC Data Source and Set Options Programmatically” on page 2-17

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

rmoptions

Package: database.options.jdbc.sqlserver

Remove JDBC connection options

Syntax

```
opts = rmoptions(opts,option)
```

Description

`opts = rmoptions(opts,option)` removes one or more specified JDBC connection options from the `SQLConnectionOptions` object `opts`.

Examples

Remove Driver-Specific Connection Option

Edit an existing JDBC data source for a Microsoft® SQL Server® database. Set an additional JDBC driver-specific option, and test the database connection. Then, remove the additional JDBC driver-specific option, and test and save the data source.

Retrieve the existing SQL Server data source `SQLServerDataSource`.

```
datasource = "SQLServerDataSource";  
opts = databaseConnectionOptions(datasource)
```

```
opts =  
  SQLConnectionOptions with properties:
```

```
      DataSourceName: "SQLServerDataSource"  
      Vendor: "Microsoft SQL Server"  
  
      JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"  
      DatabaseName: "toystore_doc"  
      Server: "dbtb04"  
      PortNumber: 54317  
      AuthenticationType: "Windows"
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `JDBCDriverLocation` — Full path of the JDBC driver file
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

- `AuthenticationType` — Authentication type

Add a JDBC driver-specific connection option by using a name-value pair argument. The option specifies a timeout value for establishing the database connection. `opts` contains a new property for the additional JDBC connection option.

```
opts = setoptions(opts, 'loginTimeout', "20")
```

```
opts =
  SqlConnectionOptions with properties:
      DataSourceName: "SQLServerDataSource"
      Vendor: "Microsoft SQL Server"
      JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
      DatabaseName: "toystore_doc"
      Server: "dbtb04"
      PortNumber: 54317
      AuthenticationType: "Windows"
  Additional Connection Options:
      loginTimeout: "20"
```

Test the database connection with a blank user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)

status = logical
      1
```

Remove the JDBC driver-specific option for specifying a timeout value. The `opts` object no longer contains the `loginTimeout` property.

```
opts = rmoptions(opts, 'loginTimeout')

opts =
  SqlConnectionOptions with properties:
      DataSourceName: "SQLServerDataSource"
      Vendor: "Microsoft SQL Server"
      JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
      DatabaseName: "toystore_doc"
      Server: "dbtb04"
      PortNumber: 54317
      AuthenticationType: "Windows"
```

Test the database connection again.

```
username = "";
password = "";
status = testConnection(opts, username, password)
```

```
status = logical  
      1
```

Save the data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, specified as an SQLConnectionOptions object.

option — JDBC connection option

character vector | string scalar | cell array of character vectors | string array

JDBC connection option, specified as a character vector, string scalar, cell array of character vectors, or string array. Specify the name of one or more JDBC connection options or driver-specific connection options.

Example: ["DatabaseName" "Server" "PortNumber"]

Example: "loginTimeout"

Data Types: char | string | cell

Output Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, returned as an SQLConnectionOptions object.

Version History

Introduced in R2020b

See Also

Objects

SQLConnectionOptions

Functions

databaseConnectionOptions | setoptions | saveAsDataSource | database | deleteDataSource | testConnection | reset

Topics

“Create JDBC Data Source and Set Options Programmatically” on page 2-17

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

saveAsDataSource

Package: database.options.jdbc.sqlserver

Save JDBC data source

Syntax

```
saveAsDataSource(opts)
```

Description

`saveAsDataSource(opts)` saves the JDBC data source specified by the `SQLConnectionOptions` object `opts`.

Examples

Create JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft® SQL Server® database.

Create an SQL Server data source for a JDBC database connection.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc", vendor)

opts =
  SQLConnectionOptions with properties:
      DataSourceName: ""
      Vendor: "Microsoft SQL Server"

      JDBCDriverLocation: ""
      DatabaseName: ""
      Server: "localhost"
      PortNumber: 1433
      AuthenticationType: "Server"
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `JDBCDriverLocation` — Full path of the JDBC driver file
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number
- `AuthenticationType` — Authentication type

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, full path to the JDBC driver file, database name `toystore_doc`, database server `dbtb04`, port number `54317`, and Windows® authentication.

```
opts = setoptions(opts, ...
    'DataSourceName', "SQLServerDataSource", ...
    'JDBCDriverLocation', "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...
    'PortNumber', 54317, 'AuthenticationType', "Windows")
```

```
opts =
    SqlConnectionOptions with properties:
```

```
        DataSourceName: "SQLServerDataSource"
        Vendor: "Microsoft SQL Server"

        JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
        DatabaseName: "toystore_doc"
        Server: "dbtb04"
        PortNumber: 54317
        AuthenticationType: "Windows"
```

The `setoptions` function sets the `DataSourceName`, `JDBCDriverLocation`, `DatabaseName`, `Server`, `PortNumber`, and `AuthenticationType` properties in the `SqlConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)
```

```
status = logical
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `database` function or the Database Explorer app.

Input Arguments

opts — Database connection options

`SqlConnectionOptions` object

Database connection options, specified as an `SqlConnectionOptions` object.

Version History

Introduced in R2020b

See Also

Objects

SQLConnectionOptions

Functions

databaseConnectionOptions | setoptions | roptions | database | deleteDataSource | testConnection | reset

Topics

“Create JDBC Data Source and Set Options Programmatically” on page 2-17

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

setoptions

Package: `database.options.jdbc.sqlserver`

Set JDBC connection options

Syntax

```
opts = setoptions(opts,Option1,OptionValue1,...,OptionN,OptionValueN)
```

Description

`opts = setoptions(opts,Option1,OptionValue1,...,OptionN,OptionValueN)` sets JDBC connection options using the `SQLConnectionOptions` object `opts`.

Examples

Create JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft® SQL Server® database.

Create an SQL Server data source for a JDBC database connection.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc",vendor)

opts =
  SQLConnectionOptions with properties:
      DataSourceName: ""
      Vendor: "Microsoft SQL Server"

      JDBCDriverLocation: ""
      DatabaseName: ""
      Server: "localhost"
      PortNumber: 1433
      AuthenticationType: "Server"
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `JDBCDriverLocation` — Full path of the JDBC driver file
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number
- `AuthenticationType` — Authentication type

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, full path to the JDBC driver file, database name `toystore_doc`, database server `dbtb04`, port number `54317`, and Windows® authentication.

```
opts = setoptions(opts, ...
    'DataSourceName', "SQLServerDataSource", ...
    'JDBCDriverLocation', "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...
    'PortNumber', 54317, 'AuthenticationType', "Windows")
```

```
opts =
    SQLConnectionOptions with properties:
```

```
        DataSourceName: "SQLServerDataSource"
        Vendor: "Microsoft SQL Server"

        JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
        DatabaseName: "toystore_doc"
        Server: "dbtb04"
        PortNumber: 54317
        AuthenticationType: "Windows"
```

The `setoptions` function sets the `DataSourceName`, `JDBCDriverLocation`, `DatabaseName`, `Server`, `PortNumber`, and `AuthenticationType` properties in the `SQLConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)
```

```
status = logical
        1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `database` function or the Database Explorer app.

Edit Existing JDBC Data Source

Edit an existing JDBC data source for a Microsoft® SQL Server® database. Set an additional JDBC driver-specific option, and save the data source.

Retrieve the existing SQL Server data source `SQLServerDataSource`.

```
datasource = "SQLServerDataSource";
opts = databaseConnectionOptions(datasource)
```

```
opts =
    SQLConnectionOptions with properties:
```

```
DataSourceName: "SQLServerDataSource"
Vendor: "Microsoft SQL Server"

JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
DatabaseName: "toystore_doc"
Server: "dbtb04"
PortNumber: 54317
AuthenticationType: "Windows"
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `JDBCDriverLocation` — Full path of the JDBC driver file
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number
- `AuthenticationType` — Authentication type

Add a JDBC driver-specific connection option by using a name-value pair argument. The option specifies a timeout value for establishing the database connection. `opts` contains a new section of properties for the additional JDBC connection option.

```
opts = setoptions(opts, 'loginTimeout', "20")
```

```
opts =
  SQLConnectionOptions with properties:

    DataSourceName: "SQLServerDataSource"
    Vendor: "Microsoft SQL Server"

    JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
    DatabaseName: "toystore_doc"
    Server: "dbtb04"
    PortNumber: 54317
    AuthenticationType: "Windows"
```

```
Additional Connection Options:
```

```
loginTimeout: "20"
```

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)

status = logical
      1
```

Save the updated data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, specified as an SQLConnectionOptions object.

Option1,OptionValue1,...,OptionN,OptionValueN — JDBC connection options to set

name-value pair arguments

JDBC connection options to set, specified as one or more name-value pair arguments. **Option** is a character vector or string scalar that specifies the name of a JDBC connection option. **OptionValue** specifies the value of the JDBC connection option. **OptionValue** can be a character vector, string scalar, logical scalar, or numeric scalar. You can specify any JDBC connection option that is a property of the SQLConnectionOptions object.

Example: "DataSourceName", "myDataSource", "Server", "localhost", "PortNumber", 3306 configures a JDBC data source named myDataSource that is located on the local server with the port number 3306.

Output Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, returned as an SQLConnectionOptions object.

Version History

Introduced in R2020b

See Also

Apps

Database Explorer

Objects

SQLConnectionOptions

Functions

deleteDataSource | databaseConnectionOptions | rmoptions | saveAsDataSource | testConnection | database | reset

Topics

“Create JDBC Data Source and Set Options Programmatically” on page 2-17

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

testConnection

Package: database.options.jdbc.sqlserver

Test JDBC database connection

Syntax

```
status = testConnection(opts,username,password)
[status,message] = testConnection(opts,username,password)
```

Description

`status = testConnection(opts,username,password)` tests the JDBC database connection specified by the `SQLConnectionOptions` object `opts`, a user name, and a password.

`[status,message] = testConnection(opts,username,password)` also returns the error message associated with testing the database connection.

Examples

Create JDBC Data Source

Create, configure, test, and save a JDBC data source for a Microsoft® SQL Server® database.

Create an SQL Server data source for a JDBC database connection.

```
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc",vendor)

opts =
    SQLConnectionOptions with properties:
        DataSourceName: ""
        Vendor: "Microsoft SQL Server"
        JDBCDriverLocation: ""
        DatabaseName: ""
        Server: "localhost"
        PortNumber: 1433
        AuthenticationType: "Server"
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `JDBCDriverLocation` — Full path of the JDBC driver file
- `DatabaseName` — Name of the database

- `Server` — Name of the database server
- `PortNumber` — Port number
- `AuthenticationType` — Authentication type

Configure the data source by setting the JDBC connection options for the data source `SQLServerDataSource`, full path to the JDBC driver file, database name `toystore_doc`, database server `dbtb04`, port number `54317`, and Windows® authentication.

```
opts = setoptions(opts, ...
    'DataSourceName', "SQLServerDataSource", ...
    'JDBCDriverLocation', "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...
    'PortNumber', 54317, 'AuthenticationType', "Windows")
```

```
opts =
    SQLConnectionOptions with properties:

        DataSourceName: "SQLServerDataSource"
        Vendor: "Microsoft SQL Server"

        JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
        DatabaseName: "toystore_doc"
        Server: "dbtb04"
        PortNumber: 54317
        AuthenticationType: "Windows"
```

The `setoptions` function sets the `DataSourceName`, `JDBCDriverLocation`, `DatabaseName`, `Server`, `PortNumber`, and `AuthenticationType` properties in the `SQLConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts,username,password)

status = logical
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `database` function or the Database Explorer app.

Retrieve Message for JDBC Database Connection Test

Create and configure a JDBC data source to a Microsoft® SQL Server® database. Test the database connection to the JDBC data source and retrieve the error message.

Create an SQL Server data source for a JDBC database connection.

```

vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc",vendor)

opts =
  SqlConnectionOptions with properties:
      DataSourceName: ""
      Vendor: "Microsoft SQL Server"

      JDBCDriverLocation: ""
      DatabaseName: ""
      Server: "localhost"
      PortNumber: 1433
      AuthenticationType: "Server"

```

opts is an SqlConnectionOptions object with these properties:

- DataSourceName — Name of the data source
- Vendor — Database vendor name
- JDBCDriverLocation — Full path of the JDBC driver file
- DatabaseName — Name of the database
- Server — Name of the database server
- PortNumber — Port number
- AuthenticationType — Authentication type

Configure the data source by setting the JDBC connection options for the data source SQLServerDataSource, full path to the JDBC driver file, database name toystore_doc, database server dbtb04, port number 54317, and SQL Server authentication.

```

opts = setoptions(opts, ...
  'DataSourceName', "SQLServerDataSource", ...
  'JDBCDriverLocation', "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar", ...
  'DatabaseName', "toystore_doc", 'Server', "dbtb04", ...
  'PortNumber', 54317, 'AuthenticationType', "Server")

```

```

opts =
  SqlConnectionOptions with properties:
      DataSourceName: "SQLServerDataSource"
      Vendor: "Microsoft SQL Server"

      JDBCDriverLocation: "C:\Drivers\mssql-jdbc-7.0.0.jre8.jar"
      DatabaseName: "toystore_doc"
      Server: "dbtb04"
      PortNumber: 54317
      AuthenticationType: "Server"

```

The setoptions function sets the DataSourceName, JDBCDriverLocation, DatabaseName, Server, PortNumber, and AuthenticationType properties in the SqlConnectionOptions object.

Test the database connection using an incorrect user name and password. The `testConnection` function returns the logical `0`, which indicates the database connection fails. Retrieve and display the error message for the failed connection.

```
username = "wronguser";
password = "wrongpassword";
[status,message] = testConnection(opts,username,password)
```

```
status = logical
        0
```

```
message =
'JDBC Driver Error: Login failed for user 'wronguser'. ClientConnectionId:dfc57aed-973f-4c5f-838'
```

Input Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, specified as an SQLConnectionOptions object.

username — User name

character vector | string scalar

User name required to access the database, specified as a character vector or string scalar. If no user name is required, specify an empty value "".

Data Types: char | string

password — Password

character vector | string scalar

Password required to access the database, specified as a character vector or string scalar. If no password is required, specify an empty value "".

Data Types: char | string

Output Arguments

status — Connection status

logical

Connection status, returned as a logical `true` if the connection test passes or `false` if the connection test fails.

message — Error message

character vector

Error message, returned as a character vector. If the connection test passes, then the error message is an empty character vector. Otherwise, the error message contains text from the failed database connection.

Version History

Introduced in R2020b

See Also

Objects

SQLConnectionOptions

Functions

databaseConnectionOptions | setoptions | roptions | saveAsDataSource | database | deleteDataSource | reset

Topics

“Create JDBC Data Source and Set Options Programmatically” on page 2-17

“Modify and Delete Data Sources” on page 4-17

“Configure Driver and Data Source” on page 2-14

connection

Relational database ODBC connection

Description

Create a database connection using an ODBC driver. For details about ODBC drivers and the alternative JDBC drivers, see “Choose Between ODBC and JDBC Drivers” on page 2-12.

You can use the `connection` object to connect to various databases using different drivers that you install and administer. For details, see “Connect to Database” on page 2-121.

Creation

Create a `connection` object using the `odbc` or `database` function.

Properties

Connection Properties

DataSource — Data source name

' ' (default) | character vector

This property is read-only.

Data source name for ODBC connection, specified as a character vector. `DataSource` is the name you provide for your data source when you create a data source using the Microsoft ODBC Administrator.

The data source name is an empty character vector when the connection is invalid.

Example: 'MS SQL Server'

Data Types: char

UserName — User name

' ' (default) | character vector

This property is read-only.

User name required to access the database, specified as a character vector. If no user name is required, specify an empty value ' '.

Example: 'username'

Data Types: char

Message — Database connection status message

' ' (default) | character vector

This property is read-only.

Database connection status message, specified as a character vector. The status message is empty when the database connection is successful. Otherwise, this property contains an error message.

Example: 'ODBC Driver Error: [Micro ...'

Data Types: char

Type — Database connection type

'ODBC Connection Object'

This property is read-only.

Database connection type, specified as the value 'ODBC Connection Object' that means a database connection created using an ODBC driver.

Data Types: char

Database Properties**AutoCommit — Flag to autocommit transactions**

'on' (default) | 'off'

Flag to autocommit transactions, specified as one of these values:

- 'on' — Database transactions are automatically committed to the database.
- 'off' — Database transactions must be committed to the database manually.

Example: 'AutoCommit', 'off'

ReadOnly — Read-only database data

'off' (default) | 'on'

Read-only database data, specified as one of these values:

- 'on' — Database data is read-only.
- 'off' — Database data is writable.

Data Types: char

LoginTimeout — Login timeout

0 (default) | positive numeric scalar

This property is read-only.

Login timeout, specified as a positive numeric scalar. The login timeout specifies the number of seconds that the driver waits while trying to connect to a database before throwing an error.

When no login timeout for the connection attempt is specified, the value is 0.

When login timeout is not supported by the database, the value is -1.

Data Types: double

MaxDatabaseConnections — Maximum database connections

-1 (default) | positive numeric scalar

This property is read-only.

Maximum database connections, specified as a positive, numeric scalar.

The value is 0 when there is no upper limit to the maximum number of database connections.

When the maximum number of database connections is not supported by the database, the value is -1.

Data Types: double

Catalog and Schema Information

DefaultCatalog — Default catalog name

' ' (default) | character vector

This property is read-only.

Default catalog name, specified as a character vector.

When a database does not specify a default catalog, the value is an empty character vector ' '.

Example: 'catalog'

Data Types: char

Catalogs — Catalog names

{ } (default) | cell array of character vectors

This property is read-only.

Catalog names, specified as a cell array of character vectors.

When a database does not contain catalogs, the value is an empty cell array { }.

Example: {'catalog1', 'catalog2'}

Data Types: cell

Schemas — Schema names

{ } (default) | cell array of character vectors

This property is read-only.

Schema names, specified as a cell array of character vectors.

When a database does not contain schemas, the value is an empty cell array { }.

Example: {'schema1', 'schema2', 'schema3'}

Data Types: cell

Database and Driver Information

DatabaseProductName — Database product name

' ' (default) | character vector

This property is read-only.

Database product name, specified as a character vector.

When the database connection is invalid, the value is an empty character vector ' '.

Example: 'Microsoft SQL Server'

Data Types: char

DatabaseProductVersion — Database product version

' ' (default) | character vector

This property is read-only.

Database product version, specified as a character vector.

When the database connection is invalid, the value is an empty character vector ' '.

Example: '11.00.2100'

Data Types: char

DriverName — Driver name

' ' (default) | character vector

This property is read-only.

Driver name of an ODBC driver, specified as a character vector.

When the database connection is invalid, the value is an empty character vector ' '.

Example: 'sqlncl11.dll'

Data Types: char

DriverVersion — Driver version

' ' (default) | character vector

This property is read-only.

Driver version of an ODBC driver, specified as a character vector.

When the database connection is invalid, the value is an empty character vector ' '.

Example: '11.00.5058'

Data Types: char

Object Functions

close	Close and invalidate database and driver resource utilizer
commit	Make database changes permanent
execute	Execute SQL statement using relational database connection
fetch	Import data into MATLAB workspace from execution of SQL statement
isopen	Determine if database connection is open
rollback	Undo database changes
executeSQLScript	Execute SQL script on database
select	Execute SQL SELECT statement and import data into MATLAB
sqlfind	Find information about all table types in database
sqlinnerjoin	Inner join between two database tables
sqlouterjoin	Outer join between two database tables
sqlread	Import data into MATLAB from database table
sqlwrite	Insert MATLAB data into database table

sqlupdate	Update rows in database table
update	Replace data in database table with MATLAB data

Examples

Connect to MySQL Using ODBC Driver

First, create an ODBC connection to the MySQL database. Then, import data from the database into MATLAB and perform simple data analysis. Close the database connection. The code assumes that you are connecting to a MySQL database version 5.5.46 using the MySQL ODBC 5.3 ANSI Driver.

Connect to the database using the data source name, user name, and password.

```
datasource = 'dsname';
username = 'username';
password = 'pwd';

conn = database(datasource,username,password)

conn =

    connection with properties:

        DataSource: 'MySQLdb'
        UserName: 'username'
        Message: ''
        Type: 'ODBC Connection Object'
    Database Properties:

        AutoCommit: 'on'
        ReadOnly: 'off'
        LoginTimeout: 0
        MaxDatabaseConnections: 0
    Catalog and Schema Information:

        DefaultCatalog: 'catalog'
        Catalogs: {'catalog1', 'catalog2'}
        Schemas: {}
    Database and Driver Information:

        DatabaseProductName: 'MySQL'
        DatabaseProductVersion: '5.5.46-0+deb7u1'
        DriverName: 'myodbc5a.dll'
        DriverVersion: '05.03.0004'
```

`conn` has an empty `Message` property, which indicates a successful connection.

The property sections of the connection object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `select` function. Display the data.

```
selectquery = 'SELECT * FROM inventoryTable';  
data = select(conn,selectquery)
```

```
ans =
```

<u>productnumber</u>	<u>quantity</u>	<u>price</u>	<u>inventorydate</u>
1	1700	14.5	'2014-09-23 09:38:34.0'
2	1200	9.3	'2014-07-08 22:50:45.0'
3	356	17.2	'2014-05-14 07:14:28.0'
...			

Determine the highest product quantity from the table.

```
max(data.Quantity)
```

```
ans =
```

```
9000
```

Close the database connection `conn`.

```
close(conn)
```

Alternative Functionality

You can connect to an SQLite database file by creating the `sqlite` object. This connection uses the MATLAB interface to SQLite that does not require installing or administering a database or driver. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Version History

Introduced before R2006a

See Also

`select` | `fetch` | `close`

Topics

“Configure Driver and Data Source” on page 2-14

“MySQL ODBC for Windows” on page 2-47

“Connect to Database” on page 2-121

“Import Data from Database Table Using `sqlread` Function” on page 5-58

“Insert Data into Database Table” on page 5-61

“Database Connection Error Messages” on page 3-7

connection

Relational database JDBC connection

Description

Create a database connection using a JDBC driver. For details about JDBC drivers and the alternative ODBC drivers, see “Choose Between ODBC and JDBC Drivers” on page 2-12.

You can use the `connection` object to connect to various databases using different drivers that you install and administer. For details, see “Connect to Database” on page 2-121.

Creation

Create a `connection` object using the `database` function.

Properties

Connection Properties

DataSource — Data source name

' ' (default) | character vector

This property is read-only.

Database name for JDBC connection, specified as a character vector. `DataSource` is the name of your database. The name differs for different database systems. For example, `DataSource` is the SID or the service name when you are connecting to an Oracle database. Or, `DataSource` is the catalog name when you are connecting to a MySQL database. For details about your database name, contact your database administrator or refer to your database documentation.

The data source name is an empty character vector when the connection is invalid.

Example: 'MS SQL Server'

Data Types: char

UserName — User name

' ' (default) | character vector

This property is read-only.

User name required to access the database, specified as a character vector. If no user name is required, specify an empty value ' '.

Example: 'username'

Data Types: char

Message — Database connection status message

' ' (default) | character vector

This property is read-only.

Database connection status message, specified as a character vector. The status message is empty when the database connection is successful. Otherwise, this property contains an error message.

Example: 'JDBC Driver Error: [Micro ...'

Data Types: char

Type — Database connection type

'JDBC Connection Object'

This property is read-only.

Database connection type, specified as the value 'JDBC Connection Object' that means a database connection created using a JDBC driver.

Data Types: char

JDBC Connection Properties

Driver — JDBC driver

' ' (default) | character vector

This property is read-only.

JDBC driver, specified as a character vector when connecting to a database using a JDBC driver URL. This property depends on the URL property.

Example: 'com.mysql.jdbc.jdbc2.opti ...'

Data Types: char

URL — Database connection URL

' ' (default) | character vector

This property is read-only.

Database connection URL, specified as a character vector for a vendor-specific string. This property depends on the Driver property.

Example: 'jdbc:mysql://sname:1234/ ...'

Data Types: char

Database Properties

AutoCommit — Flag to autocommit transactions

'on' (default) | 'off'

Flag to autocommit transactions, specified as one of these values:

- 'on' — Database transactions are automatically committed to the database.
- 'off' — Database transactions must be committed to the database manually.

Example: 'AutoCommit', 'off'

ReadOnly — Read-only database data

'off' (default) | 'on'

Read-only database data, specified as one of these values:

- 'on' — Database data is read-only.
- 'off' — Database data is writable.

Data Types: char

LoginTimeout — Login timeout

0 (default) | positive numeric scalar

This property is read-only.

Login timeout, specified as a positive numeric scalar. The login timeout specifies the number of seconds that the driver waits while trying to connect to a database before throwing an error.

When no login timeout for the connection attempt is specified, the value is 0.

When login timeout is not supported by the database, the value is -1.

Data Types: double

MaxDatabaseConnections — Maximum database connections

-1 (default) | positive numeric scalar

This property is read-only.

Maximum database connections, specified as a positive, numeric scalar.

The value is 0 when there is no upper limit to the maximum number of database connections.

When the maximum number of database connections is not supported by the database, the value is -1.

Data Types: double

Catalog and Schema Information

DefaultCatalog — Default catalog name

' ' (default) | character vector

This property is read-only.

Default catalog name, specified as a character vector.

When a database does not specify a default catalog, the value is an empty character vector ' '.

Example: 'catalog'

Data Types: char

Catalogs — Catalog names

{ } (default) | cell array of character vectors

This property is read-only.

Catalog names, specified as a cell array of character vectors.

When a database does not contain catalogs, the value is an empty cell array { }.

Example: {'catalog1', 'catalog2'}

Data Types: cell

Schemas — Schema names

{ } (default) | cell array of character vectors

This property is read-only.

Schema names, specified as a cell array of character vectors.

When a database does not contain schemas, the value is an empty cell array {}.

Example: {'schema1', 'schema2', 'schema3'}

Data Types: cell

Database and Driver Information

DatabaseProductName — Database product name

' ' (default) | character vector

This property is read-only.

Database product name, specified as a character vector.

When the database connection is invalid, the value is an empty character vector ' '.

Example: 'Microsoft SQL Server'

Data Types: char

DatabaseProductVersion — Database product version

' ' (default) | character vector

This property is read-only.

Database product version, specified as a character vector.

When the database connection is invalid, the value is an empty character vector ' '.

Example: '11.00.2100'

Data Types: char

DriverName — Driver name

' ' (default) | character vector

This property is read-only.

Driver name of a JDBC driver, specified as a character vector.

When the database connection is invalid, the value is an empty character vector ' '.

Example: 'sqlnc111.dll'

Data Types: char

DriverVersion — Driver version

' ' (default) | character vector

This property is read-only.

Driver version of a JDBC driver, specified as a character vector.

When the database connection is invalid, the value is an empty character vector ''.

Example: '11.00.5058'

Data Types: char

Object Functions

close	Close and invalidate database and driver resource utilizer
commit	Make database changes permanent
execute	Execute SQL statement using relational database connection
fetch	Import data into MATLAB workspace from execution of SQL statement
isopen	Determine if database connection is open
rollback	Undo database changes
executeSQLScript	Execute SQL script on database
select	Execute SQL SELECT statement and import data into MATLAB
sqlfind	Find information about all table types in database
sqlinnerjoin	Inner join between two database tables
sqlouterjoin	Outer join between two database tables
sqlread	Import data into MATLAB from database table
sqlwrite	Insert MATLAB data into database table
sqlupdate	Update rows in database table
update	Replace data in database table with MATLAB data
runstoredprocedure	Call stored procedure with and without input and output arguments

Examples

Connect to Oracle Using JDBC Driver

Create a JDBC connection to an Oracle database. To create this connection, you must configure a JDBC data source. For more information, see the `databaseConnectionOptions` function. Then, import data from the database into MATLAB, perform simple data analysis, and close the database connection.

This example assumes that you are connecting to an Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 using the Oracle JDBC Driver 12.1.0.1.0.

Connect to the database using a JDBC data source name, user name, and password.

```
datasource = 'dsname';
username = 'username';
password = 'pwd';
```

```
conn = database(datasource,username,password)
```

```
conn =
```

```
connection with properties:
```

```
DataSource: 'dsname'
UserName: 'username'
Driver: 'oracle.jdbc.pool.OracleDa ...'
URL: 'jdbc:oracle:thin:@(DESCRI ...'
```

```

        Message: ''
        Type: 'JDBC Connection Object'
Database Properties:
    AutoCommit: 'on'
    ReadOnly: 'off'
    LoginTimeout: 0
    MaxDatabaseConnections: 0
Catalog and Schema Information:
    DefaultCatalog: ''
    Catalogs: {}
    Schemas: {'schema1', 'schema2', 'schema3' ... and 39 more}
Database and Driver Information:
    DatabaseProductName: 'Oracle'
    DatabaseProductVersion: 'Oracle Database 12c Enter ...'
    DriverName: 'Oracle JDBC driver'
    DriverVersion: '12.1.0.1.0'

```

`conn` has an empty `Message` property, which indicates a successful connection.

The property sections of the connection object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `select` function. Display the data.

```
selectquery = 'SELECT * FROM inventoryTable';
data = select(conn,selectquery)
```

ans =

productnumber	quantity	price	inventorydate
1	1700	14.5	'2014-09-23 09:38:34.0'
2	1200	9.3	'2014-07-08 22:50:45.0'
3	356	17.2	'2014-05-14 07:14:28.0'
...			

Determine the highest product quantity in the table.

```
max(data.Quantity)
```

ans =

```
9000
```

Close the database connection.

```
close(conn)
```

Alternative Functionality

You can connect to an SQLite database file by creating the `sqlite` object. This connection uses the MATLAB interface to SQLite that does not require installing or administering a database or driver.

For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Version History

Introduced before R2006a

See Also

select | fetch | close

Topics

“Configure Driver and Data Source” on page 2-14

“Oracle JDBC for Windows” on page 2-41

“Connect to Database” on page 2-121

“Import Data from Database Table Using sqlread Function” on page 5-58

“Insert Data into Database Table” on page 5-61

“Database Connection Error Messages” on page 3-7

cursor

(Not recommended) Database cursor

Note The `cursor` object is not recommended. Use the `fetch` function instead. For details, see “Compatibility Considerations”.

The scrollable cursor functionality has no replacement.

Description

After connecting to a relational database using either ODBC or JDBC drivers, you can perform actions using the database connection. To import data into MATLAB from a database and perform database operations, you must create a `cursor` object. Database Toolbox uses this object to retrieve rows from database tables and execute SQL statements.

There are two types of database cursors, basic and scrollable. Basic cursors let you import data in an SQL query in a sequential way. However, scrollable cursors enable data import from a specified offset in the data set.

To import data quickly using a SQL `SELECT` statement, use the `select` function. To import data with full functionality, use the `exec` and `fetch` functions. For differences, see “Data Import Using Database Explorer App or Command Line” on page 2-125.

A cursor object stays open until you close it using the `close` function.

Creation

Create a cursor object using the `exec` function.

Properties

ODBC and JDBC Driver Properties

Data — SQL query results

cell array (default) | table | structure | numeric | dataset

SQL query results, specified as a cell array, table, structure, numeric, or dataset array. After running the `exec` function, this property is blank. The `fetch` function populates this property with imported data from the executed SQL query.

To set the data return format, use the `setdbprefs` function.

Note The dataset array value will be removed in a future release. Use table instead.

Example: [15×5 table]

Data Types: `double` | `struct` | `table` | `cell`

RowLimit — Number of rows to import

0 (default) | positive numeric scalar

This property is read-only.

Number of rows to import at a time, specified as a positive numeric scalar.

Data Types: `double`

SQLQuery — SQL query

character vector

This property is read-only.

SQL query, specified as a character vector. To change the SQL query, create a `cursor` object and specify the SQL query in the input argument `sqlquery` of the `exec` function.

For information about the SQL query language, see the SQL Tutorial.

Example: `'SELECT * FROM productTable'`

Data Types: `char`

Message — Error message

' ' (default) | character vector

This property is read-only.

Error message, specified as a character vector. An empty character vector specifies that the `exec` or `fetch` functions executed successfully. If this property is empty after running `exec`, then the SQL statement executed successfully. If this property is empty after running `fetch`, then the data import completed successfully. Otherwise, the property populates with the returned error message.

To throw error messages to the Command Window, use the `setdbprefs` function. Enter this code:

```
setdbprefs('ErrorHandling','report');
sqlquery = 'SELECT * FROM invalidtablename';
curs = exec(conn,sqlquery)
```

To store error messages in the `Message` property instead, enter this code:

```
setdbprefs('ErrorHandling','store');
sqlquery = 'SELECT * FROM invalidtablename';
curs = exec(conn,sqlquery)
```

Example: `'Table 'scheme.InvalidTableName' doesn't exist'`

Data Types: `char`

Type — Database cursor type

'ODBCCursor Object' | 'Database Cursor Object'

This property is read-only.

Database cursor type, specified as one of these values.

Value	Database Cursor Type
'ODBCCursor Object'	cursor object created using an ODBC database connection
'Database Cursor Object'	cursor object created using a JDBC database connection

Statement – Statement

C statement object | Java statement object

This property is read-only.

Statement, specified as a C statement object or Java statement object.

Example: [1×1 com.mysql.jdbc.StatementImpl]

Scrollable – Scrollable cursor

0 (default) | 1

This property is read-only.

Scrollable cursor, specified as a logical value. The value 0 identifies the cursor object as basic. The value 1 identifies the cursor object as scrollable.

Note This property is hidden.

Data Types: logical

Position – Cursor position

0 (default) | numeric scalar

This property is read-only.

Cursor position of a scrollable cursor in the data set, specified as a numeric scalar. Only scrollable cursors have this property. The cursor position behaves differently depending on the database driver used to establish the database connection.

Data Types: double

JDBC Driver Properties**DatabaseObject – JDBC connection**

connection object

This property is read-only.

JDBC connection, specified as a connection object created by connecting to a database using the JDBC driver.

Example: [1×1 database.jdbc.connection]

ResultSet – Result set

Java result set object

This property is read-only.

Result set, specified as a Java result set object.

Example: [1×1 com.mysql.jdbc.JDBC4ResultSet]

Cursor — Database cursor

Java object

This property is read-only.

Database cursor, specified as an internal Java object that represents the cursor object.

Example: [1×1 com.mathworks.toolbox.database.sqlExec]

Fetch — Imported data

Java object

This property is read-only.

Imported data, specified as an internal Java object that represents the imported data.

Example: [1×1 com.mathworks.toolbox.database.fetchTheData]

Object Functions

close (Not recommended) Close cursor
 fetch (Not recommended) Import data into MATLAB workspace from database cursor
 get (To be removed) Retrieve object properties
 isopen (Not recommended) Determine if database cursor is open

Examples

Select Data Using Native ODBC Driver

Use a native ODBC connection to import product data from a Microsoft SQL Server database into MATLAB. Then, determine the highest unit cost among products.

Create an ODBC database connection to a Microsoft SQL Server database with Windows authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Select all data from the table `productTable` using the connection object. Assign the SQL `SELECT` statement to the variable `sqlquery`. The cursor object contains the executed SQL query.

```
sqlquery = 'SELECT * FROM productTable';
curs = exec(conn,sqlquery)
```

```
curs =  
  
    cursor with properties:  
  
        Data: 0  
        RowLimit: 0  
        SQLQuery: 'SELECT * FROM productTable'  
        Message: []  
        Type: 'ODBCCursor Object'  
        Statement: [1x1 database.internal.ODBCStatementHandle]
```

For an ODBC connection, the `Type` property contains `ODBCCursor Object`. For JDBC connections, the `Type` property contains `Database Cursor Object`.

Import data from the table into MATLAB.

```
curs = fetch(curs);  
data = curs.Data;
```

Determine the highest unit cost in the table.

```
max(data.unitCost)
```

```
ans =
```

```
    24
```

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

Version History

Introduced before R2006a

R2018b: cursor object is not recommended

Not recommended starting in R2018b

The `cursor` object is not recommended. Use the `fetch` function instead. Some differences between the workflows might require updates to your code.

There are no plans to remove the `cursor` object at this time.

Update Code

Use the `fetch` function with the `connection` object to import data from a database in one step.

In prior releases, you wrote multiple lines of code to create the `cursor` object and import data. For example:

```
curs = exec(conn,sqlquery);  
curs = fetch(curs);  
results = curs.Data;  
close(curs)
```

Now you can import data in one step using the `fetch` function.

```
results = fetch(conn,sqlquery);
```

See Also

close | database | fetch | setdbprefs

close

Package: database.odbcc

(Not recommended) Close cursor

Note The `close` function is not recommended. There is no replacement for this functionality. To import data, use the `fetch` function. For details, see “Compatibility Considerations”.

Syntax

```
close(curs)
```

Description

`close(curs)` closes and invalidates the database cursor and driver resource utilizer to free up database and driver resources.

Examples

Close cursor Object

Connect to a Microsoft SQL Server database and verify the database connection. Then, import data from the database into MATLAB. Determine the highest unit cost among the retrieved products in the table. Close the database cursor and database connection.

Create an ODBC database connection to a Microsoft SQL Server database with Windows authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, the connection is successful.

```
conn.Message
```

```
ans =
```

```
    []
```

Select all data from the table `productTable` by using the `connection` object, and sort the data by product number. Assign the SQL `SELECT` statement to the variable `sqlquery`. The cursor object contains the executed SQL query.

```
sqlquery = 'SELECT * FROM productTable ORDER BY productNumber';
curs = exec(conn, sqlquery);
```

Import the data from the executed SQL query and display the first three rows.

```
curs = fetch(curs);
curs.Data(1:3,:)
```

```
ans =
```

```
3×5 table
```

	productNumber	stockNumber	supplierNumber	unitCost	productDescription
1		4.0035e+05	1001	14	'Building Blocks'
2		4.0031e+05	1002	9	'Painting Set'
3		4.01e+05	1009	17	'Slinky'

Determine the highest unit cost in the table.

```
data = curs.Data;
max(data.unitCost)
```

```
ans =
```

```
24
```

After you finish working with the cursor object, close it.

```
close(curs)
```

After you close the cursor object, MATLAB deletes the object. Use the `clear` function to remove the `curs` variable from the MATLAB workspace.

```
curs
clear curs
```

```
curs =
```

```
handle to deleted cursor
```

Close the database connection.

```
close(conn)
```

Input Arguments

curs — Database cursor

cursor object

Database cursor, specified as a cursor object created using the `exec` function.

Version History

Introduced before R2006a

R2018b: close function is not recommended

Not recommended starting in R2018b

The `close` function is not recommended. Use the `fetch` function to import data. Some differences between the workflows might require updates to your code.

There are no plans to remove the `close` function at this time.

Update Code

Use the `fetch` function with the `connection` object to import data from a database in one step.

In prior releases, you wrote multiple lines of code to create the `cursor` object and import data. For example:

```
curs = exec(conn,sqlquery);  
curs = fetch(curs);  
results = curs.Data;  
close(curs)
```

Now you can import data in one step using the `fetch` function.

```
results = fetch(conn,sqlquery);
```

There is no replacement functionality for the `close` function.

See Also

database | `close` | `fetch`

External Websites

SQL Tutorial

fetch

Package: database.odbcc

(Not recommended) Import data into MATLAB workspace from database cursor

Note The `fetch` function with the `cursor` object is not recommended. Use the `fetch` function with the `connection` object or the `select` function instead. For details, see “Compatibility Considerations”.

The scrollable cursor functionality has no replacement.

Syntax

```
curs = fetch(curs)
curs = fetch(curs, rowlimit)
curs = fetch( ___, Name, Value)
```

Description

`curs = fetch(curs)` imports all rows of data from an executed SQL query into the `Data` property of the cursor object. Use the cursor object to investigate imported data and its structure.

Caution: Leaving cursor and connection objects open or overwriting open objects can result in unexpected behavior. After you finish working with these objects, you must close them using `close`.

`curs = fetch(curs, rowlimit)` imports the maximum number of rows of data, as specified in `rowlimit`, from an executed SQL query.

`curs = fetch(___, Name, Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, `curs = fetch(curs, 'AbsolutePosition', 5)`; imports data using an absolute position offset in a scrollable cursor, whereas `curs = fetch(curs, 'RelativePosition', 10)`; imports data using a relative position offset.

Examples

Import All Data Using cursor Object

Import product data from a Microsoft SQL Server database into MATLAB by using the `cursor` object. Then, determine the highest unit cost among products in the table.

Create an ODBC database connection to a Microsoft SQL Server database with Windows authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the Message property is empty, the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Execute the SQL query using the exec function and the database connection. Then, import all the data from productTable.

```
sqlquery = 'SELECT * FROM productTable';
curs = exec(conn,sqlquery);
curs = fetch(curs)
```

```
curs =
```

```
 cursor with properties:
```

```
      Data: [15x5 table]
      RowLimit: 0
      SQLQuery: 'SELECT * FROM productTable'
      Message: []
      Type: 'ODBCCursor Object'
      Statement: [1x1 database.internal.ODBCStatementHandle]
```

For an ODBC connection, the Type property contains ODBC Cursor Object. For JDBC connections, this property contains Database Cursor Object.

Display the data in the cursor object property Data.

```
curs.Data
```

```
ans =
```

```
 15x5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	1.2597e+05	1003	13	'Victorian Doll'
8	2.1257e+05	1001	5	'Train Set'
7	3.8912e+05	1007	16	'Engine Kit'
2	4.0031e+05	1002	9	'Painting Set'
4	4.0034e+05	1008	21	'Space Cruiser'
1	4.0035e+05	1001	14	'Building Blocks'
5	4.0046e+05	1005	3	'Tin Soldier'
6	4.0088e+05	1004	8	'Sail Boat'
3	4.01e+05	1009	17	'Slinky'
10	8.8865e+05	1006	24	'Teddy Bear'
11	4.0814e+05	1004	11	'Convertible'
12	2.1046e+05	1010	22	'Hugsy'
13	4.7082e+05	1012	17	'Pancakes'
14	5.101e+05	1011	19	'Shawl'
15	8.9975e+05	1011	20	'Snacks'

Determine the highest unit cost in the table.

```
data = curs.Data;
max(data.unitCost)
```

```
ans =
```

```
    24
```

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)
close(conn)
```

Input Arguments

curs — Database cursor

cursor object

Database cursor, specified as a cursor object created using the `exec` function.

rowlimit — Row limit

numeric scalar

Row limit, specified as a positive numeric scalar that indicates the maximum number of rows of data to import from the database.

If `rowlimit` is 0, `fetch` returns all the rows of data.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

```
Example: curs = fetch(curs, 'RelativePosition', 10);
```

AbsolutePosition — Absolute position offset

numeric scalar

Absolute position offset, specified as the comma-separated pair consisting of `'AbsolutePosition'` and a numeric scalar that indicates the absolute position offset value. When you specify an absolute position offset value, `fetch` imports data starting from the cursor position equal to this value, regardless of the current cursor location. The scalar can be a positive number to signify fetching data from the start of the data set, or a negative number to signify fetching data from the end of the data set. This name-value pair argument is available only when you create a scrollable cursor object using `exec`.

```
Example: 'AbsolutePosition', 5
```

Data Types: `double`

RelativePosition — Relative position offset

numeric scalar

Relative position offset, specified as the comma-separated pair consisting of 'RelativePosition' and a numeric scalar that indicates the relative position offset value. When you specify a relative position offset value, `fetch` adds the current cursor position value to the relative position offset value. Then, `fetch` imports data starting from the resulting value. The scalar can be a positive number to signify importing data after the current cursor position in the data set, or a negative number to signify importing data before the current cursor position in the data set. This name-value pair argument is available only when you create a scrollable cursor object using `exec`.

Example: 'RelativePosition',10

Data Types: double

Output Arguments

curs — Database cursor

cursor object

Database cursor, returned as a cursor object populated with imported data in the `Data` property. You can specify the output data format in the `Data` property by using the `setdbprefs` function.

Tips

- If you have a native ODBC connection that you established using `database`, then running `fetch` on the cursor object updates the input cursor object itself. Depending on whether you provide an output argument, the same object gets copied over to the output. Therefore, only one cursor object exists in memory for any of these usages:

```
% First use
curs = fetch(curs)
% Second use
fetch(curs)
% Third use
curs2 = fetch(curs)
```

Version History

Introduced before R2006a

R2018b: fetch function with the cursor object is not recommended

Not recommended starting in R2018b

The `fetch` function with the cursor object is not recommended. Use the `fetch` function with the connection object instead. Some differences between the workflows might require updates to your code.

There are no plans to remove the `fetch` function at this time.

Update Code

Use the `fetch` function with the connection object to import data from a database in one step.

In prior releases, you wrote multiple lines of code to create the cursor object and import data. For example:

```
curs = exec(conn,sqlquery);
curs = fetch(curs);
```

```
results = curs.Data;  
close(curs)
```

Now you can import data in one step using the `fetch` function.

```
results = fetch(conn,sqlquery);
```

You can also import data in one step using the `select` function.

```
data = select(conn,selectquery);
```

The scrollable cursor functionality has no replacement.

See Also

`fetch` | `close` | `database` | `setdbprefs` | `exec`

External Websites

SQL Tutorial

isopen

Package: database.odbc

(Not recommended) Determine if database cursor is open

Note The `isopen` function is not recommended. There is no replacement for this functionality. To import data, use the `fetch` function. For details, see “Compatibility Considerations”.

Syntax

```
i = isopen(curs)
```

Description

`i = isopen(curs)` returns 1 if the database cursor is open and 0 if the database cursor is closed or invalid.

Examples

Determine If Database Cursor Is Open

Connect to a Microsoft SQL Server database and verify the database cursor. Then, import data from the database into MATLAB. Determine the highest unit cost among the retrieved products in the table. Close the database cursor and database connection.

Create an ODBC database connection to a Microsoft SQL Server database with Windows authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, the connection is successful.

```
conn.Message
```

```
ans =
```

```
    []
```

Create a cursor object by executing an SQL query in the database. Select all the data from the table `productTable` by using the connection object, and then sort the data by the product number.

```
sqlquery = 'SELECT * FROM productTable ORDER BY productNumber';  
curs = exec(conn, sqlquery);
```

Determine if the database cursor is open. The `isopen` function returns the numeric scalar 1, which means the database cursor is open.

```
i = isopen(curs)
```

```
i =
```

```
1
```

Import data from the executed SQL query and display the first three rows.

```
curs = fetch(curs);
curs.Data(1:3,:)
```

```
ans =
```

```
3×5 table
```

	productNumber	stockNumber	supplierNumber	unitCost	productDescription
1		4.0035e+05	1001	14	'Building Blocks'
2		4.0031e+05	1002	9	'Painting Set'
3		4.01e+05	1009	17	'Slinky'

Determine the highest unit cost in the table.

```
data = curs.Data;
max(data.unitCost)
```

```
ans =
```

```
24
```

After you finish working with the cursor object, close it.

```
close(curs)
```

Determine if the database cursor is closed. The `isopen` function returns the numeric scalar `0`, which means the database cursor is closed. If the cursor object is invalid, the `isopen` function returns the same result.

```
i = isopen(curs)
```

```
i =
```

```
0
```

Close the database connection.

```
close(conn)
```

Input Arguments

curs — Database cursor
cursor object

Database cursor, specified as a cursor object created using the `exec` function.

Version History

Introduced in R2015b

R2018b: isopen function is not recommended

Not recommended starting in R2018b

The `isopen` function is not recommended. Use the `fetch` function to import data. Some differences between the workflows might require updates to your code.

There are no plans to remove the `isopen` function at this time.

Update Code

Use the `fetch` function with the `connection` object to import data from a database in one step.

In prior releases, you wrote multiple lines of code to create the `cursor` object, determine if the `cursor` object is open, and import data. For example:

```
curs = exec(conn,sqlquery);  
i = isopen(curs);  
curs = fetch(curs);  
results = curs.Data;  
close(curs)
```

Now you can import data in one step using the `fetch` function.

```
results = fetch(conn,sqlquery);
```

There is no replacement functionality for the `isopen` function.

See Also

`database` | `fetch` | `close`

External Websites

SQL Tutorial

odbc

Create ODBC database connection

Syntax

```
conn = odbc(datasource,username,password)
conn = odbc(datasource,username,password,Name,Value)
conn = odbc(dsnless)
```

Description

`conn = odbc(datasource,username,password)` creates a database connection to an ODBC data source with a user name and password. The database connection `conn` is returned as an ODBC connection object.

`conn = odbc(datasource,username,password,Name,Value)` specifies options using one or more name-value arguments. For example, `'LoginTimeout',5` creates an ODBC connection with a login timeout of 5 seconds.

`conn = odbc(dsnless)` creates a connection to a database using a DSN-less connection string. (DSN is a data source name.)

Examples

Connect to MySQL Using ODBC Database Connection

Connect to a MySQL® database using an ODBC database connection. Then, import data from the database into MATLAB®, perform a simple data analysis, and close the database connection.

This example assumes that you are connecting to a MySQL Version 5.7.22 database using the MySQL ODBC 5.3 Driver.

Create a database connection to a MySQL database. Specify the user name and password.

```
datasource = "MySQL ODBC";
conn = odbc(datasource,"root","matlab")

conn =
  connection with properties:
      DataSource: 'MySQL ODBC'
      UserName: 'root'
      Message: ''
      Type: 'ODBC Connection Object'
  Database Properties:
      AutoCommit: 'on'
      ReadOnly: 'off'
      LoginTimeout: 0
      MaxDatabaseConnections: 0
```

Catalog and Schema Information:

```
DefaultCatalog: 'toystore_doc'
Catalogs: {'information_schema', 'detsdb', 'mysql' ... and 4 more}
Schemas: {}
```

Database and Driver Information:

```
DatabaseProductName: 'MySQL'
DatabaseProductVersion: '5.7.22'
DriverName: 'myodbc5a.dll'
DriverVersion: '05.03.0014'
```

conn has an empty `Message` property, which indicates a successful connection.

The property sections of the `conn` object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `fetch` function. Display the first three rows of data.

```
query = "SELECT * FROM inventoryTable";
data = fetch(conn,query);
head(data,3)
```

```
ans=3x4 table
   productNumber   Quantity   Price   inventoryDate
   _____   _____   _____   _____
           1           1700      14.5   {'2014-09-23 09:38:34'}
           2           1200         9   {'2014-07-08 22:50:45'}
           3            356        17   {'2014-05-14 07:14:28'}
```

Determine the highest product quantity in the table.

```
max(data.Quantity)
```

```
ans = 9000
```

Close the database connection.

```
close(conn)
```

Connect to MySQL Using ODBC Database Connection with Additional Options

Connect to a MySQL® database using an ODBC data source and a timeout value. Then, import data from the database into MATLAB®, perform a simple data analysis, and close the database connection.

This example assumes that you are connecting to a MySQL Version 5.7.22 database using the MySQL ODBC 5.3 Driver.

Create a database connection to a MySQL database using an ODBC data source. Specify the user name and password. Also, specify a timeout value of 5 seconds for connecting to the database.

```
datasource = "MySQL ODBC";
username = "root";
password = "matlab";
conn = odbc(datasource,username,password,'LoginTimeout',5)

conn =
  connection with properties:

      DataSource: 'MySQL ODBC'
      UserName: 'root'
      Message: ''
      Type: 'ODBC Connection Object'
  Database Properties:

      AutoCommit: 'on'
      ReadOnly: 'off'
      LoginTimeout: 5
      MaxDatabaseConnections: 0

  Catalog and Schema Information:

      DefaultCatalog: 'toystore_doc'
      Catalogs: {'information_schema', 'detsdb', 'mysql' ... and 4 more}
      Schemas: {}

  Database and Driver Information:

      DatabaseProductName: 'MySQL'
      DatabaseProductVersion: '5.7.22'
      DriverName: 'myodbc5a.dll'
      DriverVersion: '05.03.0014'
```

`conn` has an empty `Message` property, which indicates a successful connection.

The property sections of the `conn` object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `fetch` function. Display the first three rows of data.

```
query = "SELECT * FROM inventoryTable";
data = fetch(conn,query);
head(data,3)
```

```
ans=3x4 table
  productNumber  Quantity  Price  inventoryDate
  _____  _____  _____  _____
```

```
1           1700      14.5    {'2014-09-23 09:38:34'}
2           1200       9      {'2014-07-08 22:50:45'}
3           356       17     {'2014-05-14 07:14:28'}
```

Determine the highest product quantity in the table.

```
max(data.Quantity)
```

```
ans = 9000
```

Close the database connection.

```
close(conn)
```

Connect to MySQL Using DSN-Less Connection

Connect to a MySQL® database using a DSN-less database connection. Then, import data from the database into MATLAB®, perform a simple data analysis, and close the database connection.

This example assumes that you are connecting to a MySQL Version 5.7.22 database using the MySQL ODBC 5.3 Driver.

Create a database connection to a MySQL database. Specify the connection string.

```
dsnless = strcat("Driver={MySQL ODBC 5.3 Ansi Driver}; Server=dbtb01;", ...
    "Database=toystore_doc; UID=root; PWD=matlab");
conn = odbc(dsnless)

conn =
    connection with properties:

        DataSource: ''
        UserName: ''
        Message: ''
        Type: 'ODBC Connection Object'
    Database Properties:

        AutoCommit: 'on'
        ReadOnly: 'off'
        LoginTimeout: 0
        MaxDatabaseConnections: 0

    Catalog and Schema Information:

        DefaultCatalog: 'toystore_doc'
        Catalogs: {'information_schema', 'detsdb', 'mysql' ... and 4 more}
        Schemas: {}

    Database and Driver Information:

        DatabaseProductName: 'MySQL'
        DatabaseProductVersion: '5.7.22'
        DriverName: 'myodbc5a.dll'
```

```
DriverVersion: '05.03.0014'
```

conn has an empty `Message` property, which indicates a successful connection.

The property sections of the conn object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `fetch` function. Display the first three rows of data.

```
query = "SELECT * FROM inventoryTable";
data = fetch(conn,query);
head(data,3)
```

```
ans=3x4 table
   productNumber   Quantity   Price   inventoryDate
   _____   _____   _____   _____
           1           1700      14.5   {'2014-09-23 09:38:34'}
           2           1200         9   {'2014-07-08 22:50:45'}
           3            356        17   {'2014-05-14 07:14:28'}
```

Determine the highest product quantity in the table.

```
max(data.Quantity)
```

```
ans = 9000
```

Close the database connection.

```
close(conn)
```

Input Arguments

datasource — Data source name

character vector | string scalar

Data source name, specified as a character vector or string scalar. Specify the name of an existing data source.

Example: "myDataSource"

Data Types: char | string

username — User name

character vector | string scalar

User name required to access the database, specified as a character vector or string scalar. If no user name is required, specify an empty value "".

Data Types: char | string

password — Password

character vector | string scalar

Password required to access the database, specified as a character vector or string scalar. If no password is required, specify an empty value "".

Data Types: char | string

dsnless — DSN-less connection

character vector | string scalar

DSN-less connection string, specified as a character vector or string scalar. The connection string is specific to each database and usually contains connection parameters such as the database server name, port number, and database name. For details about the connection parameters of your database, see the database documentation.

This table shows some sample DSN-less connection strings for the Windows and Linux platforms. To use these samples, substitute your values for the corresponding connection parameters in the strings. The values might vary based on your database configuration.

Database	DSN-Less Connection String
Microsoft SQL Server	<p>Windows — "Driver={SQL Server Native Client 11.0}; Server=localhost \toy_store; Port=1433; Database=toy_store; UID=user; PWD=password"</p> <p>Linux — "Driver={ODBC Driver 17 for SQL Server}; Server=localhost,1433; Database=toy_store; UID=user; PWD=password"</p>
MySQL	<p>Windows — "Driver={MySQL ODBC 5.3 Ansi Driver}; Server=localhost; Database=toy_store; UID=user; PWD=password"</p> <p>Linux — "Driver={MySQL ODBC 5.3}; Server=localhost; Database=toy_store; UID=user; PWD=password"</p>
PostgreSQL	<p>Windows — "Driver={PostgreSQL ANSI(x64)}; Server=localhost; Database=toy_store; UID=user; PWD=password"</p> <p>Linux — "Driver={PostgreSQL ANSI}; Servername=localhost; Database=toy_store; UID=user; PWD=password"</p>

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . ,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `conn = odbc(datasource,username,password,'AutoCommit','off','ReadOnly','off')` creates a database connection to an ODBC data source with a user name and password, and specifies that database transactions must be committed to the database manually and the database data is writeable.

AutoCommit — Flag to autocommit transactions

'on' (default) | 'off'

Flag to autocommit transactions, specified as one of these values:

- 'on' — Database transactions are automatically committed to the database.
- 'off' — Database transactions must be committed to the database manually.

Example: `'AutoCommit','off'`

LoginTimeout — Login timeout

0 (default) | positive numeric scalar

Login timeout, specified as a positive numeric scalar. The login timeout specifies the number of seconds that the driver waits while trying to connect to a database before throwing an error.

To specify no login timeout for the connection attempt, set the value to 0.

When the database does not support a login timeout, the function sets this value to -1.

Example: `'LoginTimeout',5`

Data Types: double

ReadOnly — Read-only database data

'off' (default) | 'on'

Read-only database data, specified as one of these values:

- 'on' — Database data is read-only.
- 'off' — Database data is writable.

Example: `'ReadOnly','on'`

Limitations

The Linux platform does not support the following:

- `select` function
- ODBC database connection using the **Database Explorer** app

- MySQL ODBC driver 8.0 and higher

Version History

Introduced in R2021a

See Also

database | configureODBCDataSource | close | sqlread | fetch | isopen | update | sqlwrite

Topics

“Setup Requirements for Database Connection” on page 2-11

“Connection Options” on page 2-8

“Configure Driver and Data Source” on page 2-14

“Microsoft SQL Server ODBC for Windows” on page 2-23

“Connect to Database” on page 2-121

“Import Data from Database Table Using sqlread Function” on page 5-58

“Insert Data into Database Table” on page 5-61

“Retrieve Database Metadata” on page 5-64

“Database Connection Error Messages” on page 3-7

database

Connect to database

Syntax

```
conn = database(datasource,username,password)
conn = database(databasename,username,password,
Param1,ParamValue1,...,ParamN,ParamValueN)
conn = database( ____,Name,Value)
conn = database(databasename,username,password,driver,url)
```

Description

`conn = database(datasource,username,password)` creates a database connection to a data source with a user name and password. The database connection is a connection object. The data source specifies whether the database connection uses an ODBC or JDBC driver.

`conn = database(databasename,username,password, Param1,ParamValue1,...,ParamN,ParamValueN)` creates a JDBC database connection to a database name with a user name, password, and JDBC driver parameters as specified by multiple name-value pair arguments.

`conn = database(____,Name,Value)` specifies options using one or more name-value pair arguments in addition to any of the input argument combinations in previous syntaxes. For example, `conn = database(datasource,username,password,'LoginTimeout',5);` creates an ODBC or JDBC connection, as specified by the `datasource` input argument, with a login timeout of 5 seconds.

`conn = database(databasename,username,password,driver,url)` creates a JDBC database connection specified by the JDBC driver name and database connection URL.

Examples

Connect to MySQL Using ODBC Driver

Connect to a MySQL® database. Then, import data from the database into MATLAB®. Perform simple data analysis, and then close the database connection.

To create a database connection using an ODBC driver, you must configure an ODBC data source.

This example assumes that you are connecting to a MySQL Version 5.7.22 database using the MySQL Driver 5.3.

Create a database connection to the ODBC data source `MySQL ODBC`. Specify the user name and password.

```
datasource = "MySQL ODBC";
username = "username";
```

```

password = "password";
conn = database(datasource,username,password)

conn =
  connection with properties:

      DataSource: 'MySQL ODBC'
      UserName: 'username'
      Message: ''
      Type: 'ODBC Connection Object'
  Database Properties:

      AutoCommit: 'on'
      ReadOnly: 'off'
      LoginTimeout: 0
      MaxDatabaseConnections: 0

  Catalog and Schema Information:

      DefaultCatalog: 'toystore_doc'
      Catalogs: {'information_schema', 'detsdb', 'mysql' ... and 4 more}
      Schemas: {}

  Database and Driver Information:

      DatabaseProductName: 'MySQL'
      DatabaseProductVersion: '5.7.22'
      DriverName: 'myodbc5a.dll'
      DriverVersion: '05.03.0014'

```

`conn` has an empty `Message` property, which indicates a successful connection.

The property sections of the `conn` object are:

- **Database Properties** -- Information about the database configuration
- **Catalog and Schema Information** -- Names of catalogs and schemas in the database
- **Database and Driver Information** -- Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB® using the `sqlread` function. Display the first eight rows of data.

```

tablename = "inventoryTable";
data = sqlread(conn,tablename);
head(data)

```

```

ans=8x4 table
  productNumber  Quantity  Price  inventoryDate
  _____  _____  _____  _____
      1           1700     14.5  {'2014-09-23 09:38:34'}
      2           1200      9      {'2014-07-08 22:50:45'}
      3           356     17      {'2014-05-14 07:14:28'}
      4          2580     21      {'2013-06-08 14:24:33'}
      5          9000      3      {'2012-09-14 15:00:25'}
      6          4540      8      {'2013-12-25 19:45:00'}
      7          6034     16      {'2014-08-06 08:38:00'}

```

```
8           8350           5           {'2011-06-18 11:45:35'}
```

Determine the highest product quantity in the table.

```
max(data.Quantity)
```

```
ans = 9000
```

Close the database connection.

```
close(conn)
```

Connect to PostgreSQL Using JDBC Driver URL

Connect to the PostgreSQL database. Then, import data from the database into MATLAB, perform simple data analysis, and then close the database connection. This example assumes that you are connecting to a PostgreSQL 9.4.5 database using the JDBC PostgreSQL Native Driver 8.4.

Connect to the database using the database name, user name, and password. Use the JDBC driver `org.postgresql.Driver` to make the connection.

Use the URL defined by the driver vendor including your server name host, port number, and database name.

```
databasename = 'dbname';
username = 'username';
password = 'pwd';
driver = 'org.postgresql.Driver';
url = 'jdbc:postgresql://host:port/dbname';

conn = database(databasename,username,password,driver,url)
```

```
conn =
  connection with properties:
      DataSource: 'dbname'
      UserName: 'username'
      Driver: 'org.postgresql.Driver'
      URL: 'jdbc:postgresql://host: ...'
      Message: ''
      Type: 'JDBC Connection Object'
  Database Properties:
      AutoCommit: 'on'
      ReadOnly: 'off'
      LoginTimeout: 0
      MaxDatabaseConnections: 8192
  Catalog and Schema Information:
      DefaultCatalog: 'catalog'
      Catalogs: {'catalog'}
      Schemas: {'schema1', 'schema2', 'schema3' ... and 1 more}
  Database and Driver Information:
      DatabaseProductName: 'PostgreSQL'
      DatabaseProductVersion: '9.4.5'
      DriverName: 'PostgreSQL Native Driver'
      DriverVersion: 'PostgreSQL 8.4 JDBC4 (bui ...'
```

`conn` has an empty `Message` property, which indicates a successful connection.

The property sections of the conn object are:

- Database Properties — Information about the database configuration
- Catalog and Schema Information — Names of catalogs and schemas in the database
- Database and Driver Information — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `select` function. Display the data.

```
selectquery = 'SELECT * FROM inventoryTable';
data = select(conn,selectquery)
```

```
ans =
```

productnumber	quantity	price	inventorydate
1	1700	14.5	'2014-09-23 09:38:34.0'
2	1200	9.3	'2014-07-08 22:50:45.0'
3	356	17.2	'2014-05-14 07:14:28.0'
...			

Determine the highest quantity in the table.

```
max(data.quantity)
```

```
ans =
```

```
9000
```

Close the database connection.

```
close(conn)
```

Connect to MySQL Using ODBC Driver with Additional Options

Connect to the MySQL® database using an ODBC driver. Then, import data from the database into MATLAB®, perform simple data analysis, and then close the database connection. The example assumes that you are connecting to the MySQL database version 5.7.22 and MySQL ODBC 5.3 ANSI driver.

Create a database connection to a MySQL database and a login timeout of 5 seconds. Specify the user name and password.

```
databasename = "toystore_doc";
username = "username";
password = "password";
conn = database(databasename,username,password,'Vendor','MySQL', ...
    'Server','dbtb01','PortNumber',3306,'LoginTimeout',5)
```

```
conn =
```

```
connection with properties:
```

```
DataSource: 'toystore_doc'
UserName: 'username'
```

```

        Driver: 'com.mysql.cj.jdbc.Driver'
        URL: 'jdbc:mysql://dbtb01:3306/ ...'
        Message: ''
        Type: 'JDBC Connection Object'
Database Properties:
        AutoCommit: 'on'
        ReadOnly: 'off'
        LoginTimeout: 5
        MaxDatabaseConnections: 0

Catalog and Schema Information:
        DefaultCatalog: 'toystore_doc'
        Catalogs: {'detsdb', 'information_schema', 'mysql' ... and 4 more}
        Schemas: {}

Database and Driver Information:
        DatabaseProductName: 'MySQL'
        DatabaseProductVersion: '5.7.22'
        DriverName: 'MySQL Connector/J'
        DriverVersion: 'mysql-connector-java-8.0. ...'

```

`conn` has an empty `Message` property, which indicates a successful connection.

The property sections of the `conn` object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `select` function. Display the first three rows of data.

```

selectquery = "SELECT * FROM inventoryTable";
data = select(conn,selectquery);
head(data,3)

```

```

ans=3x4 table
    productNumber    Quantity    Price    inventoryDate
    _____    _____    _____    _____
         1           1700        14.5    {'2014-09-23 09:38:34'}
         2           1200         9      {'2014-07-08 22:50:45'}
         3            356         17    {'2014-05-14 07:14:28'}

```

Determine the highest quantity in the table.

```
max(data.Quantity)
```

```
ans = 9000
```

Close the database connection.

```
close(conn)
```

Input Arguments

datasource — Data source name

character vector | string scalar

Data source name, specified as a character vector or string scalar. Specify the name of an existing data source.

Example: "myDataSource"

Data Types: char | string

databasename — JDBC database name

character vector | string scalar

JDBC database name, specified as a character vector or string scalar. Specify the name of your database to create a database connection using a JDBC driver.

The name differs for different database systems. For example, `databasename` is the SID or the service name when you are connecting to an Oracle database. Or, `databasename` is the catalog name when you are connecting to a MySQL database.

For details about your database name, contact your database administrator or refer to your database documentation.

Data Types: char | string

username — User name

character vector | string scalar

User name required to access the database, specified as a character vector or string scalar. If no user name is required, specify an empty value "".

Data Types: char | string

password — Password

character vector | string scalar

Password required to access the database, specified as a character vector or string scalar. If no password is required, specify an empty value "".

Data Types: char | string

Param1, ParamValue1, ..., ParamN, ParamValueN — JDBC driver parameters

name-value pair arguments

JDBC driver parameters, specified as multiple name-value pair arguments. A `Param` argument is a character vector or string scalar that specifies the name of a JDBC driver parameter. A `ParamValue` argument is a character vector, string scalar, or numeric scalar that specifies the value of the JDBC driver parameter.

Param Valid Values	Param Value Description	ParamValue Valid Values
"Vendor"	Database vendor	<ul style="list-style-type: none"> 'MySQL' 'Oracle' 'Microsoft SQL Server' 'PostgreSQL' <p>If you are connecting to a database system not listed here, use the driver and url syntax.</p>
"Server"	Database server name or address	<ul style="list-style-type: none"> character vector string scalar 'localhost' (default)
"PortNumber"	Server port number where the server is listening	Numeric scalar
"AuthType"	Authentication type (required only for Microsoft SQL Server)	<ul style="list-style-type: none"> 'Server' — Microsoft SQL Server authentication 'Windows' — Windows authentication
"DriverType"	Driver type (required only for Oracle)	<ul style="list-style-type: none"> 'thin' — Thin driver 'oci' — Windows authentication

Tip: When creating a JDBC connection using the JDBC driver parameters, you can omit the following:

- 'Server' parameter when connecting to a database locally
- 'PortNumber' parameter when connecting to a database server listening on the default port (except for Oracle connections)

Example: `'Vendor','Microsoft SQL Server','Server','dbtb04','AuthType','Windows','PortNumber',54317` connects to a Microsoft SQL Server database using a JDBC driver on a machine named `dbtb04` with Windows authentication and using port number 54317.

Example: `'Vendor','MySQL','Server','remotehost'` connects to a MySQL database using a JDBC driver on a machine named `remotehost`.

driver — JDBC driver name

character vector | string scalar

JDBC driver name, specified as a character vector or string scalar that refers to the name of the Java driver that implements the `java.sql.Driver` interface. For details, see JDBC driver name and database connection URL on page 11-125.

Data Types: `char` | `string`

url — Database connection URL

character vector | string scalar

Database connection URL, specified as a character vector or string scalar for the vendor-specific URL. This URL is typically constructed using connection properties such as server name, port number, and database name. For details, see JDBC driver name and database connection URL on page 11-125. If you do not know the driver name or the URL, you can use name-value pair arguments to specify individual connection properties.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `'LoginTimeout',5,'ErrorHandling','report'` specifies waiting for 5 seconds to connect to a database before throwing an error and displaying any error messages at the command line.

AutoCommit — Flag to autocommit transactions

`'on'` (default) | `'off'`

Flag to autocommit transactions, specified as one of these values:

- `'on'` — Database transactions are automatically committed to the database.
- `'off'` — Database transactions must be committed to the database manually.

Example: `'AutoCommit','off'`

LoginTimeout — Login timeout

0 (default) | positive numeric scalar

Login timeout, specified as the comma-separated pair consisting of `'LoginTimeout'` and a positive numeric scalar. The login timeout specifies the number of seconds that the driver waits while trying to connect to a database before throwing an error.

To specify no login timeout for the connection attempt, set the value to 0.

When login timeout is unsupported by the database, the value is -1.

Example: 'LoginTimeout',5

Data Types: double

ReadOnly — Read-only database data

'off' (default) | 'on'

Read-only database data, specified as the comma-separated pair consisting of 'ReadOnly' and one of these values:

- 'on' — Database data is read-only.
- 'off' — Database data is writable.

Example: 'ReadOnly','on'

ErrorHandling — Error handling

'store' (default) | 'report'

Error handling, specified as the comma-separated pair consisting of 'ErrorHandling' and one of these values:

- 'store' — Store an error message in the Message property of the connection object.
- 'report' — Display an error message at the command line.

Output Arguments

conn — Database connection

connection object

Database connection, returned as an ODBC connection object or JDBC connection object.

More About

JDBC Driver Name and Database Connection URL

The JDBC driver name and database connection URL take different forms for different databases. For details, consult your database driver documentation.

Database	JDBC Driver Name and Database URL Example Syntax
IBM® Informix®	<p>JDBC driver: com.informix.jdbc.IfxDriver</p> <p>Database URL: jdbc:informix-sqli://161.144.202.206:3000:INFORMIXSERVER=stars</p>
Microsoft SQL Server 2005	<p>JDBC driver: com.microsoft.sqlserver.jdbc.SQLServerDriver</p> <p>Database URL: jdbc:sqlserver://localhost:port;database=databasename</p>

Database	JDBC Driver Name and Database URL Example Syntax
MySQL	<p>JDBC driver: <code>twz1.jdbc.mysql.jdbcMySQLDriver</code></p> <p>Database URL: <code>jdbc:z1MySQL://natasha:3306/metrics</code></p> <p>For MySQL Connector 8.0 and later:</p> <p>JDBC driver: <code>com.mysql.cj.jdbc.Driver</code></p> <p>For previous versions of MySQL Connector:</p> <p>JDBC driver: <code>com.mysql.jdbc.Driver</code></p> <p>Database URL: <code>jdbc:mysql://devmetrics.mrkps.com/testing</code></p> <p>To insert or select characters with encodings that are not default, append the value <code>useUnicode=true&characterEncoding=<i>encoding</i></code> to the URL, where <i>encoding</i> is any valid MySQL character encoding followed by <code>&</code>. For example, <code>useUnicode=true&characterEncoding=utf8&</code>.</p> <p><i>The trailing & is required.</i></p>
Oracle oci7 drivers	<p>JDBC driver: <code>oracle.jdbc.driver.OracleDriver</code></p> <p>Database URL: <code>jdbc:oracle:oci7:@rex</code></p>
Oracle oci8 drivers	<p>JDBC driver: <code>oracle.jdbc.driver.OracleDriver</code></p> <p>Database URL: <code>jdbc:oracle:oci8:@111.222.333.44:1521:</code></p> <p>Database URL: <code>jdbc:oracle:oci8:@frug</code></p>
Oracle 10 Connections with JDBC (Thin drivers)	<p>JDBC driver: <code>oracle.jdbc.driver.OracleDriver</code></p> <p>Database URL: <code>jdbc:oracle:thin:</code></p>
Oracle Thin drivers	<p>JDBC driver: <code>oracle.jdbc.driver.OracleDriver</code></p> <p>Database URL: <code>jdbc:oracle:thin:@144.212.123.24:1822:</code></p> <p>Database URL: <code>jdbc:oracle:thin:@(DESCRIPTION = (ADDRESS = (PROTOCOL = TCP)(HOST = ServerName)(PORT = 1234)) (CONNECT_DATA = (SERVER = DEDICATED) (SERVICE_NAME = dbname)))</code></p>
PostgreSQL	<p>JDBC driver: <code>org.postgresql.Driver</code></p> <p>Database URL: <code>jdbc:postgresql://host:port/database</code></p>
PostgreSQL with SSL Connection	<p>JDBC driver: <code>org.postgresql.Driver</code></p> <p>Database URL: <code>jdbc:postgresql:servername:dbname:ssl=true&sslfactory=org.postgresql.ssl.NonValidatingFactory&</code></p> <p><i>The trailing & is required.</i></p>
Teradata®	<p>JDBC driver: <code>com.teradata.jdbc.TeraDriver</code></p> <p>Database URL: <code>jdbc:teradata://DatabaseServerName</code></p>

Tips

- If you specify a data source name in the `datasource` input argument that appears on both ODBC and JDBC data source lists, then the `database` function creates an ODBC database connection. In this case, if you must create a JDBC database connection instead, append `_JDBC` to the name of the data source.

Alternative Functionality

Database Explorer App

The `database` function connects to a database using the command line. To connect to a database and explore its data in a visual way, use the **Database Explorer** app.

Version History

Introduced before R2006a

R2020b: `database.ODBCConnection` syntax has been removed

Errors starting in R2020b

The `database.ODBCConnection` syntax has been removed. Use the syntaxes of the `database` function instead. Some differences between the workflows require updates to your code.

Update Code

In prior releases, you created a connection to a database by using the `database.ODBCConnection` syntax. For example:

```
conn = database.ODBCConnection(datasource,username,password);
```

Now use the `database` syntax instead.

```
conn = database(datasource,username,password);
```

See Also

Functions

`close` | `sqlread` | `fetch` | `isopen` | `update` | `sqlwrite`

Apps

Database Explorer

Topics

“Setup Requirements for Database Connection” on page 2-11

“Connection Options” on page 2-8

“Configure Driver and Data Source” on page 2-14

“Microsoft SQL Server ODBC for Windows” on page 2-23

“Microsoft SQL Server JDBC for Windows” on page 2-29

“PostgreSQL JDBC for Windows” on page 2-61

“Connect to Database” on page 2-121

“Import Data from Database Table Using `sqlread` Function” on page 5-58

“Insert Data into Database Table” on page 5-61

“Retrieve Database Metadata” on page 5-64

“Database Connection Error Messages” on page 3-7

datastore

(Not recommended) Create datastore to access collection of data in a database

Note datastore is not recommended. Use `databaseDatastore` instead.

Syntax

```
dbds = datastore(conn,sqlquery)
```

Description

This `datastore` function creates a `DatabaseDatastore` object. You can use this object to read large volumes of data in a relational database.

A `DatabaseDatastore` is one of the available datastore types. You can create other types of datastores using the MATLAB function `datastore`. After creating any datastore, you can analyze data by writing custom functions to run MapReduce using the `mapreduce` function. For details, see “Getting Started with MapReduce”.

`dbds = datastore(conn,sqlquery)` creates a `DatabaseDatastore` object `dbds` using the database connection `conn`. This datastore contains query results from the executed SQL query `sqlquery`.

Examples

Create a DatabaseDatastore

Create a database connection `conn` using the ODBC driver. This code assumes that you are connecting to a MySQL database with the data source named `MySQL`, user name `username`, and password `pwd`. `MySQL` contains the table named `productTable` with 15 product records.

```
conn = database('MySQL','username','pwd');
```

Create a `DatabaseDatastore` object `dbds` using the database connection `conn` and SQL query `sqlquery`. This SQL query retrieves all products from the product table `productTable`.

```
sqlquery = 'SELECT * FROM productTable';
```

```
dbds = datastore(conn,sqlquery)
```

```
dbds =
```

```
DatabaseDatastore with properties:
```

```
Connection: [1x1 database.odbc.connection]
Query: 'SELECT * FROM productTable'
VariableNames: {1x5 cell}
ReadSize: 10000
```

`datastore` executes the SQL query `sqlquery` and creates a cursor object with the resulting data. `dbds` contains these properties:

- connection object
- Executed SQL query
- Column names of the executed SQL query
- Number of rows to read from the SQL query results

Display the database connection property `Connection`.

```
dbds.Connection
```

```
ans =
```

```
connection with properties:
```

```
DataSource: 'MySQLdb'  
UserName: 'username'  
Message: ''  
Type: 'ODBC Connection Object'
```

```
Database Properties:
```

```
AutoCommit: 'on'  
ReadOnly: 'off'  
LoginTimeout: 0  
MaxDatabaseConnections: 0
```

```
Catalog and Schema Information:
```

```
DefaultCatalog: 'toy_store'  
Catalogs: {'information_schema', 'toy_store'}  
Schemas: {}
```

```
Database and Driver Information:
```

```
DatabaseProductName: 'MySQL'  
DatabaseProductVersion: '5.5.46-0+deb7u1'  
DriverName: 'myodbc5a.dll'  
DriverVersion: '05.03.0004'
```

The `Message` property is blank when the database connection is successful.

Close the `DatabaseDatastore` and database connection.

```
close(dbds)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the database function.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar.

For information about the SQL query language, see the SQL Tutorial.

Example: `SELECT * FROM invoice` selects all columns and rows from the `invoice` table.

Data Types: `char` | `string`

Output Arguments

dbds — Datastore containing data in database

DatabaseDatastore object

Datastore containing data in database, returned as a DatabaseDatastore object.

Version History

Introduced in R2014b

See Also

`databaseDatastore` | `database` | `datastore` | `close` | `preview` | `read`

Topics

“Import Large Data Using DatabaseDatastore Object” on page 5-23

DatabaseDatastore

“Getting Started with Datastore”

hasdata

Package: matlab.io.datastore

Determine if data in DatabaseDatastore is available to read

Syntax

```
tf = hasdata(dbds)
```

Description

`tf = hasdata(dbds)` returns logical 1 (`true`) if there is data available to read from the DatabaseDatastore object `dbds`. Otherwise, it returns logical 0 (`false`).

Examples

Determine If DatabaseDatastore Object Contains Data

Create a database connection using a JDBC driver. To create this connection, you must configure a JDBC data source. For more information, see the `configureJDBCDataSource` function. Then, create a DatabaseDatastore object and read the data stored in the object until no more data remains.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
username = "";
password = "";
conn = database(datasource,username,password);
```

Create a DatabaseDatastore object using the database connection and an SQL query. This SQL query reads the first 30 rows of data from the `airlinesmall` table.

```
sqlquery = 'select top 30 * from airlinesmall';
```

```
dbds = databaseDatastore(conn,sqlquery);
```

Read the first 10 rows.

```
dbds.ReadSize = 10;
read(dbds)
```

```
ans =
```

```
10×29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
------	-------	------------	-----------	---------	------------	---------	------------

1987	10	30	5	1329	1329	1434	1436
1987	11	7	6	1316	1315	1713	1647
1987	11	28	6	815	815	1015	1015
1987	11	2	1	700	700	800	800
1987	11	14	6	840	840	1127	1120
1987	11	1	7	1625	1625	1823	1758
1987	11	26	4	1314	1315	1538	1542
1990	10	12	5	1755	1733	1858	1820
1990	10	19	5	1130	1120	1203	1154
1990	10	8	1	1515	1440	1609	1535

Determine if the DatabaseDatastore object has additional data.

```
hasdata(dbds)
```

```
ans =
```

```
logical
```

```
1
```

When more data is available in dbds, hasdata returns 1.

Read the rest of the data in dbds, 10 rows at a time.

```
while(hasdata(dbds))
  read(dbds)
end
```

```
ans =
```

```
10x29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1990	10	23	2	1057	1055	1205	1155
1990	10	27	6	1353	1355	1634	1640
1990	9	11	2	1810	1812	1939	1930
1992	7	18	6	1538	1540	1703	1720
1992	7	19	7	932	932	1130	1052
1992	8	4	2	NaN	1815	NaN	1940
2001	12	9	7	656	650	824	823
2001	12	22	6	1707	1715	1823	1821
2001	12	8	6	1402	1410	1642	1626
2001	12	23	7	1327	1310	1530	1530

```
ans =
```

```
10x29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
------	-------	------------	-----------	---------	------------	---------	------------

2001	11	18	7	1407	1415	1442	1457
2001	11	26	1	2105	2110	2209	2237
2001	11	22	4	1345	1355	1530	1549
2002	7	25	4	1032	1035	1853	1852
2002	6	5	3	2032	2031	2233	2248
2002	6	26	3	2026	1840	47	2257
2002	6	8	6	1557	1600	1703	1711
2002	6	15	6	1120	1115	1401	1413
2002	6	10	1	927	930	1031	1031
2002	6	19	3	632	640	748	756

When no more data remains in `dbds`, `hasdata` returns logical `0` and the `while` loop stops.

Close the `DatabaseDatastore` object and the database connection.

```
close(dbds)
```

Input Arguments

dbds — Datastore containing data in database

`DatabaseDatastore` object

Datastore containing data in a database, specified as a `DatabaseDatastore` object created using the `databaseDatastore` function.

Version History

Introduced in R2014b

See Also

`database` | `databaseDatastore` | `read` | `close`

Topics

“Import Large Data Using `DatabaseDatastore` Object” on page 5-23

“Analyze Large Data in Database Using Tall Arrays” on page 5-30

“Analyze Large Data in Database Using MapReduce” on page 5-27

`DatabaseDatastore`

isopen

Package: database.odbc

Determine if database connection is open

Syntax

```
i = isopen(conn)
```

Description

`i = isopen(conn)` returns 1 if the database connection is open and 0 if the database connection is closed or invalid.

Examples

Determine If Database Connection Is Open

Connect to a Microsoft® SQL Server® database and verify the database connection. Then, import data from the database into MATLAB®. Determine the highest unit cost among the retrieved products in the table. Close the database connection.

Create an ODBC database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, the connection is successful.

```
conn.Message
```

```
ans =
```

```
    []
```

Determine if the database connection is open. The `isopen` function returns the numeric scalar 1, which means the database connection is open.

```
i = isopen(conn)
```

```
i =
```

```
    1
```

Select all the data from `productTable` and sort it by the product number. `data` is a table containing the imported data that results from executing the SQL `SELECT` statement.

```
selectquery = 'SELECT * FROM productTable ORDER BY productNumber';  
data = select(conn,selectquery);
```

Display the first three rows of data.

```
data(1:3, :)
```

```
ans =
```

```
3x5 table
```

<u>productNumber</u>	<u>stockNumber</u>	<u>supplierNumber</u>	<u>unitCost</u>	<u>productDescription</u>
1	4.0035e+05	1001	14	'Building Blocks'
2	4.0031e+05	1002	9	'Painting Set'
3	4.01e+05	1009	17	'Slinky'

Determine the highest unit cost in the table.

```
max(data.unitCost)
```

```
ans =
```

```
24
```

Close the database connection.

```
close(conn)
```

Determine if the database connection is closed. The `isopen` function returns the numeric scalar `0`, which means the database connection is closed. If the database connection is invalid, the `isopen` function returns the same result.

```
i = isopen(conn)
```

```
i =
```

```
0
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

Version History

Introduced in R2015b

See Also

database | close | fetch

Topics

“Import Data from Database Table Using sqlread Function” on page 5-58

“Insert Data into Database Table” on page 5-61

“Connect to Database” on page 2-121

isopen

Determine if SQLite connection is open

Syntax

```
i = isopen(conn)
```

Description

`i = isopen(conn)` uses the MATLAB interface to SQLite to return 1 if the SQLite database connection is open and 0 if it is closed or invalid.

Examples

Determine If Database Connection Is Open

Connect to an SQLite database and verify the database connection. Then, import data from the database into MATLAB®. Determine the highest unit cost among the retrieved products in the table. Close the database connection.

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Determine if the database connection is open. The `isopen` function returns the numeric scalar 1, which means the database connection is open.

```
i = isopen(conn)
```

```
i = logical
     1
```

Select all the data from `productTable` and sort it by the product number. `data` is a table containing the imported data that results from executing the SQL `SELECT` statement.

```
sqlquery = "SELECT * FROM productTable ORDER BY productNumber";
data = fetch(conn,sqlquery);
```

Display the first three rows of data.

```
head(data,3)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
1	400345	1001	14	"Building Blocks"
2	400314	1002	9	"Painting Set"
3	400999	1009	17	"Slinky"

Determine the highest unit cost in the table.

```
max(data.unitCost)
```

```
ans = int64  
    24
```

Close the database connection.

```
close(conn)
```

Determine if the database connection is closed. The `isopen` function returns the numeric scalar `0`, which means the database connection is closed. If the database connection is invalid, the `isopen` function returns the same result.

```
i = isopen(conn)
```

```
i = logical  
    0
```

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

Version History

Introduced in R2022a

See Also

Objects

`sqlite`

Functions

`fetch` | `sqlread` | `close`

Topics

“Import Data Using MATLAB Interface to SQLite” on page 5-32

“Insert Data into SQLite Database Table” on page 5-36

preview

Package: matlab.io.datastore

Return subset of data from DatabaseDatastore

Syntax

```
data = preview(dbds)
```

Description

`data = preview(dbds)` returns the first eight rows of data from the DatabaseDatastore object `dbds` without changing its current position.

Note `preview` returns data as a table only. `preview` ignores database preference settings for data return formatting.

If there is no data to read from the query, `preview` throws an error.

Examples

Preview Data in DatabaseDatastore Object

Create a database connection using a JDBC driver. To create this connection, you must configure a JDBC data source. For more information, see the `configureJDBCDataSource` function. Then, create a DatabaseDatastore object and preview the data stored in the object.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";  
username = "";  
password = "";  
conn = database(datasource,username,password);
```

Create a DatabaseDatastore object using the database connection and an SQL query. This SQL query reads all data from the `airlinesmall` table.

```
sqlquery = 'select * from airlinesmall';  
dbds = databaseDatastore(conn,sqlquery);
```

Preview the first eight records in the data set returned by executing the SQL query in the DatabaseDatastore object.

```
preview(dbds)
```

```
ans =
```


8×29 table

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1990	9	11	2	1810	1812	1939	1930
1990	10	27	6	1353	1355	1634	1640
1990	10	23	2	1057	1055	1205	1155
1990	10	8	1	1515	1440	1609	1535
1990	10	19	5	1130	1120	1203	1154
1990	10	12	5	1755	1733	1858	1820
2001	11	22	4	1345	1355	1530	1549
2001	11	26	1	2105	2110	2209	2237

Close the DatabaseDatastore object and the database connection.

```
close(dbds)
```

Input Arguments

dbds — Datastore containing data in database

DatabaseDatastore object

Datastore containing data in a database, specified as a DatabaseDatastore object created using the databaseDatastore function.

Output Arguments

data — Query results

table

Query results, returned as a table of the first eight records in the data set. Executing the SQL statement specified in the Query property of the DatabaseDatastore object creates the data set.

If there is no data to read from the query, preview throws an error.

Version History

Introduced in R2014b

See Also

database | databaseDatastore | read | close

Topics

“Import Large Data Using DatabaseDatastore Object” on page 5-23

“Analyze Large Data in Database Using Tall Arrays” on page 5-30

“Analyze Large Data in Database Using MapReduce” on page 5-27

DatabaseDatastore

read

Package: matlab.io.datastore

Read data in DatabaseDatastore

Syntax

```
data = read(dbds)
[data,info] = read(dbds)
```

Description

`data = read(dbds)` returns data from the `DatabaseDatastore` object in increments specified by the `ReadSize` property of the `DatabaseDatastore` object. Subsequent calls to the `read` function continue reading from the endpoint of the previous call.

Note `read` returns data as a table only. `read` ignores database preference settings for data return formatting.

If there is no more data to read from the query, `read` throws an error.

`[data,info] = read(dbds)` returns database information `info`.

Note The syntax `data = read(dbds, rowcount)` has been removed. Set the `DatabaseDatastore` property `ReadSize` instead.

Examples

Read Data

Create a database connection using a JDBC driver. To create this connection, you must configure a JDBC data source. For more information, see the `configureJDBCDataSource` function. Then, create a `DatabaseDatastore` object and read the data stored in the object.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
username = "";
password = "";
conn = database(datasource,username,password);
```

Create a `DatabaseDatastore` object using the database connection and an SQL query. This SQL query retrieves all data from the `airlinesmall` table. Specify reading a maximum of 10 records from the executed SQL query.

```
sqlquery = 'select * from airlinesmall';
dbds = databaseDatastore(conn,sqlquery,'ReadSize',10);
```

Read the data in the DatabaseDatastore object.

```
data = read(dbds)
```

```
data =
```

```
10×29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	30	5	1329	1329	1434	1436
1987	11	7	6	1316	1315	1713	1647
1987	11	28	6	815	815	1015	1015
1987	11	2	1	700	700	800	800
1987	11	14	6	840	840	1127	1120
1987	11	1	7	1625	1625	1823	1758
1987	11	26	4	1314	1315	1538	1542
1987	10	28	3	1140	1140	1212	1215
1987	10	9	5	1155	1155	1250	1300
1987	10	22	4	715	715	807	803

data contains the query results.

Close the DatabaseDatastore object and the database connection.

```
close(dbds)
```

Read Data and Retrieve Database Information

Create a database connection using a JDBC driver. To create this connection, you must configure a JDBC data source. For more information, see the `configureJDBCDataSource` function. Then, create a `DatabaseDatastore` object, read the data stored in the object, and retrieve information about the database.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
username = "";
password = "";
conn = database(datasource,username,password);
```

Create a `DatabaseDatastore` object using the database connection and an SQL query. This SQL query retrieves all data from the `airlinesmall` table. Specify reading a maximum of 10 records from the executed SQL query.

```
sqlquery = 'select * from airlinesmall';
dbds = databaseDatastore(conn,sqlquery,'ReadSize',10);
```

Read the data in the DatabaseDatastore object, and retrieve information about the database.

```
[data,info] = read(dbds)
```

```
data =
```

```
10×29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	30	5	1329	1329	1434	1436
1987	11	7	6	1316	1315	1713	1647
1987	11	28	6	815	815	1015	1015
1987	11	2	1	700	700	800	800
1987	11	14	6	840	840	1127	1120
1987	11	1	7	1625	1625	1823	1758
1987	11	26	4	1314	1315	1538	1542
1987	10	28	3	1140	1140	1212	1215
1987	10	9	5	1155	1155	1250	1300
1987	10	22	4	715	715	807	803

```
info =
```

```
struct with fields:
```

```
datasource: 'MSSQLServerJDBCAuth'
offset: 10
```

`data` contains the query results. The structure `info` contains the data source name `datasource` and current cursor position `offset`.

Close the DatabaseDatastore object and the database connection.

```
close(dbds)
```

Input Arguments

dbds — Datastore containing data in database

DatabaseDatastore object

Datastore containing data in a database, specified as a DatabaseDatastore object created using the databaseDatastore function.

Output Arguments

data — Query results

table

Query results, returned as a table of the records in the data set. Executing the SQL statement specified in the `Query` property of the `DatabaseDatastore` object creates the data set. The `ReadSize` property of the `DatabaseDatastore` object specifies the number of rows in the table.

If there is no more data to read from the query, `read` throws an error.

info — Database information

structure

Database information, returned as a structure with these fields.

Field	Description
<code>datasource</code>	Data source name for ODBC drivers or a database name for JDBC drivers
<code>offset</code>	Current cursor position in the returned data set

Version History

Introduced in R2014b

See Also

`database` | `databaseDatastore` | `hasdata` | `close`

Topics

“Import Large Data Using `DatabaseDatastore` Object” on page 5-23

“Analyze Large Data in Database Using Tall Arrays” on page 5-30

“Analyze Large Data in Database Using MapReduce” on page 5-27

`DatabaseDatastore`

readall

Package: matlab.io.datastore

Read all data in DatabaseDatastore

Syntax

```
data = readall(dbds)
```

Description

`data = readall(dbds)` returns all the data in the DatabaseDatastore object `dbds`, and resets the DatabaseDatastore to the point where no data has been read from it.

Examples

Read All Data in DatabaseDatastore Object

Create a database connection using a JDBC driver. To create this connection, you must configure a JDBC data source. For more information, see the `configureJDBCDataSource` function. Then, create a DatabaseDatastore object and read all data stored in the object.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";  
username = "";  
password = "";  
conn = database(datasource,username,password);
```

Create a DatabaseDatastore object using the database connection and an SQL query. This SQL query reads all data from the `airlinesmall` table.

```
sqlquery = 'select * from airlinesmall';  
dbds = databaseDatastore(conn,sqlquery);
```

Read all data in the DatabaseDatastore object.

```
data = readall(dbds);
```

`data` contains the query results.

Display the first three rows of query results.

```
data(1:3,:)
```

```
ans =
```

3x29 table

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	28	3	1140	1140	1212	1215
1987	10	9	5	1155	1155	1250	1300
1987	10	22	4	715	715	807	803

Close the DatabaseDatastore object and the database connection.

```
close(dbds)
```

Input Arguments

dbds — Datastore containing data in database

DatabaseDatastore object

Datastore containing data in a database, specified as a DatabaseDatastore object created using the databaseDatastore function.

Output Arguments

data — Query results

table

Query results, returned as a table of the records in the data set. Executing the SQL statement specified in the Query property of the DatabaseDatastore object creates the data set.

Version History

Introduced in R2014b

See Also

database | databaseDatastore | preview | read | close

Topics

“Import Large Data Using DatabaseDatastore Object” on page 5-23

“Analyze Large Data in Database Using Tall Arrays” on page 5-30

“Analyze Large Data in Database Using MapReduce” on page 5-27

DatabaseDatastore

reset

Package: matlab.io.datastore

Reset DatabaseDatastore to initial state

Syntax

```
reset(dbds)
```

Description

`reset(dbds)` resets the `DatabaseDatastore` object `dbds` to the state where no data has been read from it. Resetting allows you to reread from the same `DatabaseDatastore`.

Examples

Reset DatabaseDatastore Object to Initial State

Create a database connection using a JDBC driver. To create this connection, you must configure a JDBC data source. For more information, see the `configureJDBCDataSource` function. Then, create a `DatabaseDatastore` object, read the data stored in the object, and reset the object to its original state.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";  
username = "";  
password = "";  
conn = database(datasource,username,password);
```

Create a `DatabaseDatastore` object using the database connection and an SQL query. This SQL query retrieves all data from the `airlinesmall` table. Specify reading a maximum of 10 records from the executed SQL query.

```
sqlquery = 'select * from airlinesmall';  
dbds = databaseDatastore(conn,sqlquery,'ReadSize',10);
```

Read data from the start of the data set.

```
read(dbds)
```

```
ans =
```

```
10×29 table
```

Year	Month	DayOfMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
------	-------	------------	-----------	---------	------------	---------	------------

1987	10	30	5	1329	1329	1434	1436
1987	11	7	6	1316	1315	1713	1647
1987	11	28	6	815	815	1015	1015
1987	11	2	1	700	700	800	800
1987	11	14	6	840	840	1127	1120
1987	11	1	7	1625	1625	1823	1758
1987	11	26	4	1314	1315	1538	1542
1987	10	28	3	1140	1140	1212	1215
1987	10	9	5	1155	1155	1250	1300
1987	10	22	4	715	715	807	803

`read` returns the first 10 records in the data set.

Reset the `DatabaseDatastore` object to its original state, where no data has been read from it. Resetting allows you to reread from the same `DatabaseDatastore` object.

```
reset(dbds)
```

Read data from the start of the data set.

```
read(dbds)
```

```
ans =
```

```
10×29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	30	5	1329	1329	1434	1436
1987	11	7	6	1316	1315	1713	1647
1987	11	28	6	815	815	1015	1015
1987	11	2	1	700	700	800	800
1987	11	14	6	840	840	1127	1120
1987	11	1	7	1625	1625	1823	1758
1987	11	26	4	1314	1315	1538	1542
1987	10	28	3	1140	1140	1212	1215
1987	10	9	5	1155	1155	1250	1300
1987	10	22	4	715	715	807	803

`read` again returns the first 10 records in the data set.

Close the `DatabaseDatastore` object and the database connection.

```
close(dbds)
```

Input Arguments

dbds — Datastore containing data in database

`DatabaseDatastore` object

Datastore containing data in a database, specified as a `DatabaseDatastore` object created using the `databaseDatastore` function.

Version History

Introduced in R2014b

See Also

database | databaseDatastore | exec | read | close

Topics

“Import Large Data Using DatabaseDatastore Object” on page 5-23

“Analyze Large Data in Database Using Tall Arrays” on page 5-30

“Analyze Large Data in Database Using MapReduce” on page 5-27

DatabaseDatastore

databaseDatastore

Datastore for data in database

Description

MATLAB has various datastores that let you import large data sets into MATLAB for analysis. A `DatabaseDatastore` object is a type of datastore that contains data from a database table or the results from executing an SQL query in a relational database. For details about other datastores, see “Getting Started with Datastore”.

With a `DatabaseDatastore` object, you can preview and read records or chunks in a data set and reset the `DatabaseDatastore` to its initial state. Also, you can analyze a large data set in a database using tall arrays or MapReduce.

Reading data from `DatabaseDatastore` objects is the same as executing the `fetch` function on the data set. Using `DatabaseDatastore` objects provides advantages that enable you to:

- Work with databases containing large amounts of data.
- Analyze large amounts of data using tall arrays with common MATLAB functions, such as `mean` and `histogram`. Create a tall array using the `tall` function. For details, see “Tall Arrays for Out-of-Memory Data”.
- Write MapReduce algorithms that define the chunking and reduction of large amounts of data by using the `mapreduce` function. For details, see “Getting Started with MapReduce”. For an example, see “Analyze Large Data in Database Using MapReduce” on page 5-27. For more MapReduce examples, see Building Effective Algorithms with MapReduce.

Creation

Syntax

```
dbds = databaseDatastore(conn,source)
dbds = databaseDatastore(conn,source,Name,Value)

dbds = databaseDatastore(conn,source,opts)
dbds = databaseDatastore(conn,source,opts,Name,Value)
```

Description

`dbds = databaseDatastore(conn,source)` creates a `DatabaseDatastore` object using the database connection. This datastore contains data from a database table or the results from an executed SQL query.

`dbds = databaseDatastore(conn,source,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'ReadSize',100` retrieves 100 rows of data from the `DatabaseDatastore` object.

`dbds = databaseDatastore(conn,source,opts)` customizes the options for importing a large data set from a database using the `SQLImportOptions` object.

`dbds = databaseDatastore(conn, source, opts, Name, Value)` specifies additional options using one or more name-value pair arguments. For example, `'Catalog', 'toy_store'` retrieves data from the `toy_store` database catalog.

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with the `database` function, connection object created with the `mysql` function, connection object created with the `postgresql` function, or `sqlite` object.

source — Source

character vector | string scalar

Source, specified as a character vector or string scalar. The source indicates whether the `DatabaseDatastore` object stores data from a database table or the results from an executed SQL query.

Example: `'inventorytable'`

Example: `"SELECT productnumber,productname FROM producttable"`

Data Types: `char` | `string`

opts — Database import options

`SQLImportOptions` object

Database import options, specified as an `SQLImportOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `databaseDatastore(conn, source, 'ReadSize', 100, 'Catalog', 'toy_store')` creates a `DatabaseDatastore` object and stores 100 rows of data from a table or SQL query using the `toy_store` database catalog.

ReadSize — Number of rows to return

numeric scalar

Number of rows to return, specified as the comma-separated pair consisting of `'ReadSize'` and a positive numeric scalar. Use this name-value pair argument to limit the number of rows for retrieval from the `DatabaseDatastore` object.

Example: `1000`

Data Types: `double`

Catalog — Database catalog name

character vector | string scalar

Database catalog name, specified as the comma-separated pair consisting of 'Catalog' and a character vector or string scalar. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have numerous catalogs.

Use the 'Catalog' name-value pair argument only when source is a database table.

Example: 'Catalog', 'toy_store'

Data Types: char | string

Schema — Database schema name

character vector | string scalar

Database schema name, specified as the comma-separated pair consisting of 'Schema' and a character vector or string scalar. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Use the 'Schema' name-value pair argument only when source is a database table.

Example: 'Schema', 'dbo'

Data Types: char | string

Limitations

- The DatabaseDatastore object supports only Microsoft SQL Server 2012 and later versions.
- The DatabaseDatastore object does not support using a parallel pool with Parallel Computing Toolbox™ installed. To analyze data using tall arrays or to run MapReduce algorithms, set the global execution environment to be the local MATLAB session by using `mapreducer`. Enter this code:

```
mapreducer(0)
```

For details about controlling parallel resources, see “Run mapreduce on a Parallel Pool” (Parallel Computing Toolbox).

- If you set the `VariableNamingRule` name-value argument to the value "modify":
 - The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the table data type.
 - The length of each variable name must be less than the number returned by `namelengthmax`.

Properties

Connection — Database connection

connection object

This property is read-only.

Database connection, specified as a connection object created using `database`.

Query — SQL query

character vector

This property is read-only.

SQL query, specified as a character vector that specifies the SQL query to execute in the database.

Data Types: char

VariableNames — Column names of retrieved data table

cell array of character vectors

Column names of the retrieved data table, specified as a cell array of one or more character vectors.

Data Types: char

SelectedVariableNames — Subset of variables to import

character vector | cell array of character vectors | numeric array

Subset of variables to import, specified as a character vector, cell array of character vectors, or numeric array that contains indices. Use the `SelectedVariableNames` property to determine the database columns to import into the MATLAB workspace.

The values in the `SelectedVariableNames` property must be equal to the values in the `VariableNames` property or a subset of these values. By default, the `SelectedVariableNames` property contains all variable names specified in the `VariableNames` property. When the `SelectedVariableNames` property specifies all variable names, the import functions of the `DatabaseDatastore` object import all database columns.

Data Types: double | char | cell

VariableNamingRule — Variable naming rule

"modify" (default) | "preserve"

Variable naming rule, specified as one of these values:

- "modify" — Remove non-ASCII characters from variable names when the `DatabaseDatastore` function imports data.
- "preserve" — Preserve most variable names when the `DatabaseDatastore` function imports data. For details, see "Limitations" on page 11-153.

If you are using the MySQL or PostgreSQL native interface, "preserve" is the default value.

The `VariableNamingRule` property has these limitations:

- The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the `table` data type.
- The length of each variable name must be less than the number returned by `namelengthmax`.

Data Types: string

ReadSize — Number of rows to read

10000 (default) | numeric scalar

Number of rows to read from the retrieved data table, specified as a nonnegative numeric scalar. To specify the number of rows to read, set the `ReadSize` property.

Example: `dbds.ReadSize = 5000;`

Data Types: double

Object Functions

hasdata	Determine if data in DatabaseDatastore is available to read
preview	Return subset of data from DatabaseDatastore
read	Read data in DatabaseDatastore
readall	Read all data in DatabaseDatastore
reset	Reset DatabaseDatastore to initial state
close	Close and invalidate database and driver resource utilizer
isPartitionable	Determine whether datastore is partitionable
isShuffleable	Determine whether datastore is shuffleable

Examples

Create DatabaseDatastore Object Using SQL Query Results

Create a database connection to a MySQL (R) database using an ODBC driver. Then, create a DatabaseDatastore object using the results from an SQL query and preview a large data set.

Create a database connection to the ODBC data source MySQL ODBC. Specify the user name and password.

```
datasource = "MySQL ODBC";
username = "username";
password = "password";
conn = database(datasource,username,password);
```

Create a DatabaseDatastore object using a database connection and an SQL query. This SQL query retrieves all flight data from the airlinesmall table. databaseDatastore executes the SQL query.

```
sqlquery = 'select * from airlinesmall';
dbds = databaseDatastore(conn,sqlquery)
```

```
dbds =
```

```
DatabaseDatastore with properties:
```

```

    Connection: [1x1 database.odbc.connection]
    Query: 'select * from airlinesmall'
    VariableNames: {1x29 cell}
    SelectedVariableNames: {1x29 cell}
    VariableNamingRule: 'modify'
    ReadSize: 10000
```

dbds is a DatabaseDatastore object with these properties:

- Connection -- Database connection object
- Query -- Executed SQL query
- VariableNames -- List of column names from the executed SQL query
- ReadSize -- Maximum number of records to read from the executed SQL query

Display the database connection property.

```
dbds.Connection
```

```
ans =
```

```
connection with properties:
```

```
    DataSource: 'MySQL ODBC'
    UserName: 'root'
    Message: ''
    Type: 'ODBC Connection Object'
```

```
Database Properties:
```

```
    AutoCommit: 'on'
    ReadOnly: 'off'
    LoginTimeout: 0
    MaxDatabaseConnections: 0
```

```
Catalog and Schema Information:
```

```
    DefaultCatalog: 'toy_store'
    Catalogs: {'information_schema', 'mysql', 'performance_schema' ... and 3 more}
    Schemas: {}
```

```
Database and Driver Information:
```

```
    DatabaseProductName: 'MySQL'
    DatabaseProductVersion: '8.0.3-rc-log'
    DriverName: 'myodbc8a.dll'
    DriverVersion: '08.00.0016'
```

The Message property is blank when the database connection is successful.

Preview the first eight records in the large data set returned by executing the SQL query in the DatabaseDatastore object.

```
preview(dbds)
```

```
ans =
```

```
8x29 table
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1990	9	22	6	1801	1750	2005	1938
1990	9	11	2	908	910	1613	1554
1990	9	2	7	NaN	1805	NaN	1900
1990	9	29	6	1434	1435	1615	1630
1990	9	3	1	925	755	1258	1144
1990	9	22	6	900	900	1241	1222
1990	9	20	4	1338	1335	1853	1907
1990	9	3	1	710	711	837	847

Close the DatabaseDatastore object and the database connection.


```
close(dbds)
```

Create DatabaseDatastore Object Using Database Table

Retrieve a large data set from a database table by creating a DatabaseDatastore object. This example uses a MySQL® database.

Create a database connection to a MySQL database with the user name and password.

```
datasource = "MySQL ODBC";
username = "username";
password = "password";
conn = database(datasource,username,password);
```

Load flight information in the MATLAB® workspace.

```
flights = readtable('airlinesmall_subset.xlsx');
```

Create the flights database table using the flight information.

```
tablename = 'flights';
sqlwrite(conn,tablename,flights)
```

Create a DatabaseDatastore object using a database connection and the flights database table.

```
dbds = databaseDatastore(conn,tablename)
```

```
dbds =
  DatabaseDatastore with properties:
        Connection: [1x1 database.odbc.connection]
        Query: 'SELECT * from flights'
  VariableNames: {1x29 cell}
SelectedVariableNames: {1x29 cell}
  VariableNamingRule: 'modify'
        ReadSize: 10000
```

dbds is a DatabaseDatastore object with these properties:

- **Connection** — Database connection object
- **Query** — Executed SQL query
- **VariableNames** — List of column names from the executed SQL query
- **ReadSize** — Maximum number of records to read from the executed SQL query

Display the database connection property.

```
dbds.Connection
```

```
ans =
  connection with properties:
        DataSource: 'MySQL ODBC'
        UserName: 'root'
        Message: ''
```

```

                                Type: 'ODBC Connection Object'
Database Properties:
    AutoCommit: 'on'
    ReadOnly: 'off'
    LoginTimeout: 0
    MaxDatabaseConnections: 0

Catalog and Schema Information:
    DefaultCatalog: 'toy_store'
    Catalogs: {'information_schema', 'mysql', 'performance_schema' ... and 3 more}
    Schemas: {}

Database and Driver Information:
    DatabaseProductName: 'MySQL'
    DatabaseProductVersion: '8.0.3-rc-log'
    DriverName: 'myodbc8a.dll'
    DriverVersion: '08.00.0016'

```

The Message property is blank when the database connection is successful.

Preview the first eight records in the data set returned by executing the SQL query in the DatabaseDatastore object.

```
preview(dbds)
```

```
ans=8x29 table
  Year   Month   DayOfMonth   DayOfWeek   DepTime   CRSDepTime   ArrTime   CRSArrTime
  -----
1996     1       18           4           2117      2120         2305      2259
1996     1       12           5           1252      1245         1511      1500
1996     1       16           2           1441      1445         1708      1721
1996     1        1           1           2258      2300         2336      2335
1996     1        4           4           1814      1814         1901      1910
1996     1       31           3           1822      1820         1934      1925
1996     1       18           4            729        730          841       843
1996     1       26           5           1704      1705         1829      1839

```

Close the DatabaseDatastore object and the database connection.

```
close(dbds)
```

Create DatabaseDatastore Object with Specific Record Count

Create a database connection using an ODBC driver. Then, create a DatabaseDatastore object by setting the ReadSize property, and preview a large data set.

Create a database connection to the ODBC data source MySQL ODBC. Specify the user name and password.

```

datasource = "MySQL ODBC";
username = "username";
password = "password";
conn = database(datasource,username,password);

```

Create a `DatabaseDatastore` object using a database connection and an SQL query. This SQL query retrieves all flight data from the `airlinesmall` table. Specify reading a maximum of 1000 records from the executed SQL query. `databaseDatastore` executes the SQL query.

```

sqlquery = 'select * from airlinesmall';

dbds = databaseDatastore(conn,sqlquery,'ReadSize',1000)

```

```
dbds =
```

```
DatabaseDatastore with properties:
```

```

        Connection: [1x1 database.odbc.connection]
        Query: 'select * from airlinesmall'
        VariableNames: {1x29 cell}
        SelectedVariableNames: {1x29 cell}
        VariableNamingRule: 'modify'
        ReadSize: 1000

```

`dbds` is a `DatabaseDatastore` object with these properties:

- `Connection` -- Database connection object
- `Query` -- Executed SQL query
- `VariableNames` -- List of column names from the executed SQL query
- `ReadSize` -- Maximum number of records to read from the executed SQL query

Display the database connection property.

```
dbds.Connection
```

```
ans =
```

```
connection with properties:
```

```

        DataSource: 'MySQL ODBC'
        UserName: 'root'
        Message: ''
        Type: 'ODBC Connection Object'

```

```
Database Properties:
```

```

        AutoCommit: 'on'
        ReadOnly: 'off'
        LoginTimeout: 0
        MaxDatabaseConnections: 0

```

```
Catalog and Schema Information:
```

```

        DefaultCatalog: 'toy_store'
        Catalogs: {'information_schema', 'mysql', 'performance_schema' ... and 3 more

```

```
Schemas: {}
```

```
Database and Driver Information:
```

```
DatabaseProductName: 'MySQL'
DatabaseProductVersion: '8.0.3-rc-log'
DriverName: 'myodbc8a.dll'
DriverVersion: '08.00.0016'
```

The Message property is blank when the database connection is successful.

Preview the first eight records in the large data set returned by executing the SQL query in the DatabaseDatastore object.

```
preview(dbds)
```

```
ans =
```

```
8x29 table
```

Year	Month	DayOfMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1990	9	22	6	1801	1750	2005	1938
1990	9	11	2	908	910	1613	1554
1990	9	2	7	NaN	1805	NaN	1900
1990	9	29	6	1434	1435	1615	1630
1990	9	3	1	925	755	1258	1144
1990	9	22	6	900	900	1241	1222
1990	9	20	4	1338	1335	1853	1907
1990	9	3	1	710	711	837	847

Close the DatabaseDatastore object and the database connection.

```
close(dbds)
```

Create DatabaseDatastore Object Using Custom Import Options

Customize import options when importing a large data set from a database table. Control the import options by creating an SQLImportOptions object. Then, customize the import options for database columns that contain logical data. Import and preview the data by creating a DatabaseDatastore object and using the preview function.

This example uses the airlinesmall_subset.xls spreadsheet, which contains the column Cancelled. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load flight information into the MATLAB® workspace.

```
flights = readtable('airlinesmall_subset.xlsx');
```

Create the flights database table using the flight information.

```
tablename = 'flights';
sqlwrite(conn,tablename,flights)
```

Create an SQLImportOptions object using the flights database table with the databaseImportOptions function.

```
opts = databaseImportOptions(conn,tablename);
```

Retrieve the default import options for the Cancelled variable.

```
varnames = 'Cancelled';
varOpts = getoptions(opts,varnames)
```

```
varOpts =
  SQLVariableImportOptions with properties:
```

```
  Variable Properties :
      Name: 'Cancelled'
      Type: 'double'
  FillValue: NaN
```

Set the import options for the data type of the specified variable to logical. Also, set the import options to replace missing data in the specified variable with the fill value true.

```
opts = setoptions(opts,varnames,'Type','logical', ...
  'FillValue',true);
```

Create the DatabaseDatastore object to import a large data set using the import options.

```
dbds = databaseDatastore(conn,tablename,opts);
```

Import the logical data in the selected variable and display a preview of the data. The imported data shows that the variable has the logical data type.

```
opts.SelectedVariableNames = varnames;
data = preview(dbds);
cancelled = data.Cancelled
```

```
cancelled = 8×1 logical array
```

```
0
0
0
0
0
0
0
0
```

Delete the flights database table using the execute function.

```
sqlquery = ['DROP TABLE ' tablename];  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Create DatabaseDatastore Object Using Custom Import Options and Database Catalog and Schema

Customize import options when importing a large data set from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options for database columns that contain logical data. Create a `DatabaseDatastore` object using the specified database catalog and schema. Import the database data and preview it by using the `preview` function with the `DatabaseDatastore` object.

This example uses the `airlinesmall_subset.xls` spreadsheet, which contains the column `Cancelled`. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Load flight information into the MATLAB® workspace.

```
flights = readtable('airlinesmall_subset.xlsx');
```

Create the `flights` database table using the flight information and the `toy_store` database catalog and `dbo` database schema.

```
tablename = 'flights';  
sqlwrite(conn,tablename,flights, ...  
         'Catalog','toy_store','Schema','dbo')
```

Create an `SQLImportOptions` object using the `flights` database table and the `databaseImportOptions` function. Specify the `toy_store` database catalog and `dbo` database schema.

```
opts = databaseImportOptions(conn,tablename, ...  
                             'Catalog','toy_store','Schema','dbo');
```

Retrieve the default import options for the `Cancelled` variable.

```
varnames = 'Cancelled';  
varOpts = getoptions(opts,varnames)
```

```
varOpts =  
    SQLVariableImportOptions with properties:
```

```
    Variable Properties :  
        Name: 'Cancelled'  
        Type: 'double'
```

```
FillValue: NaN
```

Set the import options for the data type of the specified variable to `logical`. Also, set the import options to replace missing data in the specified variable with the fill value `true`.

```
opts = setoptions(opts,varnames,'Type','logical', ...
    'FillValue',true);
```

Create the `DatabaseDatastore` object to import a large data set using import options, the `toy_store` database catalog, and the `dbo` database schema.

```
dbds = databaseDatastore(conn,tablename,opts, ...
    'Catalog','toy_store','Schema','dbo');
```

Import the logical data in the selected variable and display a preview of the data. The imported data shows that the variable has the `logical` data type.

```
opts.SelectedVariableNames = varnames;
data = preview(dbds);
cancelled = data.Cancelled
```

```
cancelled = 8×1 logical array
```

```
0
0
0
0
0
0
0
0
0
```

Delete the `flights` database table from the `toy_store` database catalog and the `dbo` database schema by using the `execute` function.

```
sqlquery = ['DROP TABLE toy_store.dbo.' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Version History

Introduced in R2014b

See Also

`database` | `sqlread` | `fetch` | `databaseImportOptions` | `setoptions` | `getoptions` | `execute` | `reset` | `preview` | `mysql`

Topics

“Import Large Data Using `DatabaseDatastore` Object” on page 5-23

“Analyze Large Data in Database Using Tall Arrays” on page 5-30

“Analyze Large Data in Database Using MapReduce” on page 5-27

“Getting Started with Datastore”

“Getting Started with MapReduce”

“Customize Options for Importing Data from Database into MATLAB” on page 5-67

“Importing Data Common Errors” on page 3-3

External Websites

SQL Tutorial

SQLPreparedStatement

SQL prepared statement

Description

The `SQLPreparedStatement` object enables you to create an SQL prepared statement. An SQL prepared statement can import, update, insert, or delete data in a database. Also, an SQL prepared statement can call stored procedures in a database.

The SQL statement can be one of these statements:

- SELECT
- INSERT
- UPDATE
- DELETE
- CALL

An SQL prepared statement contains parameters that are bound to values. By binding the parameters, you can execute the same SQL statement for different values repeatedly. The benefits of using SQL prepared statements include improved performance and security.

Note If you use SQL prepared statements with a JDBC driver and a database other than Microsoft SQL Server or PostgreSQL, the behavior of the SQL prepared statement varies based on the JDBC driver implementation of the statement. The behavior can cause unexpected results.

Creation

Create an `SQLPreparedStatement` object with the `databasePreparedStatement` function.

Properties

SQLQuery — SQL prepared statement

string scalar

This property is read-only.

SQL prepared statement query, specified as a string scalar.

Example: "SELECT * FROM inventoryTable WHERE inventoryDate > ? AND inventoryDate < ?"

Data Types: string

ParameterCount — Parameter count

numeric scalar

This property is read-only.

Parameter count, specified as a numeric scalar for the total number of parameters in the SQL prepared statement.

Data Types: double

ParameterTypes — Parameter types

string array

This property is read-only.

Parameter types, specified as a string array. The bind values must be one of the parameter types.

The parameter type is one of these data type values:

- "double"
- "string"
- "datetime"
- "logical"

Example: ["string" "string"]

Data Types: string

ParameterValues — Parameter values

cell array

Parameter values, specified as a cell array of the values to bind with the defined parameters in the SQL prepared statement.

Example: {2 5}

Data Types: cell

Object Functions

`bindParamValues` Bind values to parameters
`close` Close SQL prepared statement

Examples

Import Data Using SQL Prepared Statement

Create an SQL prepared statement to import data from a Microsoft® SQL Server® database using a JDBC database connection. Use the SELECT SQL statement for the SQL query. Import the data from the database and display the results.

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MSSQLServerJDBCAuth';  
conn = database(datasource, '', '');
```

Create an SQL prepared statement for importing data from the SQL Server database using the JDBC database connection. The question marks in the SELECT SQL statement indicate it is an SQL

prepared statement. This statement selects all data from the database table `inventoryTable` for the inventory that has an inventory date within a specified date range.

```
query = strcat("SELECT * FROM inventoryTable ", ...
    "WHERE inventoryDate > ? AND inventoryDate < ?");
pstmt = databasePreparedStatement(conn,query)
```

```
pstmt =
    SQLPreparedStatement with properties:
        SQLQuery: "SELECT * FROM inventoryTable where inventoryDate > ? AND inventoryDate < ?"
        ParameterCount: 2
        ParameterTypes: ["string" "string"]
        ParameterValues: {[] []}
```

`pstmt` is an `SQLPreparedStatement` object with these properties:

- `SQLQuery` — SQL prepared statement query
- `ParameterCount` — Parameter count
- `ParameterTypes` — Parameter types
- `ParameterValues` — Parameter values

Bind parameter values in the SQL prepared statement. Select both parameters in the SQL prepared statement using their numeric indices. Specify the values to bind as the inventory date range between January 1, 2014, and December 31, 2014. Match the format of dates in the database. The `bindParamValues` function updates the values in the `ParameterValues` property of the `pstmt` object.

```
selection = [1 2];
values = {"2014-01-01 00:00:00.000", ...
    "2014-12-31 00:00:00.000"};
pstmt = bindParamValues(pstmt,selection,values)

pstmt =
    SQLPreparedStatement with properties:
        SQLQuery: "SELECT * FROM inventoryTable where inventoryDate > ? AND inventoryDate < ?"
        ParameterCount: 2
        ParameterTypes: ["string" "string"]
        ParameterValues: {"2014-01-01 00:00:00.000"} {"2014-12-31 00:00:00.000"}
```

Import data from the database using the `fetch` function and bound parameter values. The results contain four rows of data that represent all inventory with an inventory date between January 1, 2014 and December 31, 2014.

```
results = fetch(conn,pstmt)
```

```
results=4x4 table
  productNumber  Quantity  Price  inventoryDate
-----
           1      1700    14.5  {'2014-09-23 09:38:34'}
           2      1200     9     {'2014-07-08 22:50:45'}
           3       356    17     {'2014-05-14 07:14:28'}
```

```
7          6034          16          {'2014-08-06 08:38:00'}
```

Close the SQL prepared statement and database connection.

```
close(pstmt)
close(conn)
```

Limitations

- The `SQLPreparedStatement` object supports a JDBC database connection only.

Version History

Introduced in R2019b

See Also

`database` | `close` | `fetch`

Topics

“Import Data Using SQL Prepared Statement with Multiple Parameter Values” on page 5-75
“SQL Prepared Statement Error Messages” on page 3-15

databasePreparedStatement

Create SQL prepared statement

Syntax

```
pstmt = databasePreparedStatement(conn, query)
```

Description

`pstmt = databasePreparedStatement(conn, query)` creates an `SQLPreparedStatement` object using the database connection and SQL query.

Examples

Import Data Using SQL Prepared Statement

Create an SQL prepared statement to import data from a Microsoft® SQL Server® database using a JDBC database connection. Use the `SELECT` SQL statement for the SQL query. Import the data from the database and display the results.

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MSSQLServerJDBCAuth';
conn = database(datasource, '', '');
```

Create an SQL prepared statement for importing data from the SQL Server database using the JDBC database connection. The question marks in the `SELECT` SQL statement indicate it is an SQL prepared statement. This statement selects all data from the database table `inventoryTable` for the inventory that has an inventory date within a specified date range.

```
query = strcat("SELECT * FROM inventoryTable ", ...
    "WHERE inventoryDate > ? AND inventoryDate < ?");
pstmt = databasePreparedStatement(conn, query)
```

```
pstmt =
    SQLPreparedStatement with properties:
```

```
        SQLQuery: "SELECT * FROM inventoryTable where inventoryDate > ? AND inventoryDate < ?"
        ParameterCount: 2
        ParameterTypes: ["string" "string"]
        ParameterValues: {[] []}
```

`pstmt` is an `SQLPreparedStatement` object with these properties:

- `SQLQuery` — SQL prepared statement query
- `ParameterCount` — Parameter count
- `ParameterTypes` — Parameter types

- `ParameterValues` — Parameter values

Bind parameter values in the SQL prepared statement. Select both parameters in the SQL prepared statement using their numeric indices. Specify the values to bind as the inventory date range between January 1, 2014, and December 31, 2014. Match the format of dates in the database. The `bindParamValues` function updates the values in the `ParameterValues` property of the `pstmt` object.

```
selection = [1 2];
values = {"2014-01-01 00:00:00.000", ...
         "2014-12-31 00:00:00.000"};
pstmt = bindParamValues(pstmt,selection,values)
```

```
pstmt =
```

```
SQLPreparedStatement with properties:
```

```
    SQLQuery: "SELECT * FROM inventoryTable where inventoryDate > ? AND inventoryDate < ?"
    ParameterCount: 2
    ParameterTypes: ["string" "string"]
    ParameterValues: [{"2014-01-01 00:00:00.000"} ["2014-12-31 00:00:00.000"]}
```

Import data from the database using the `fetch` function and bound parameter values. The results contain four rows of data that represent all inventory with an inventory date between January 1, 2014 and December 31, 2014.

```
results = fetch(conn,pstmt)
```

```
results=4x4 table
    productNumber    Quantity    Price    inventoryDate
    _____    _____    _____    _____
         1             1700         14.5    {'2014-09-23 09:38:34'}
         2             1200          9       {'2014-07-08 22:50:45'}
         3              356          17       {'2014-05-14 07:14:28'}
         7             6034          16       {'2014-08-06 08:38:00'}
```

Close the SQL prepared statement and database connection.

```
close(pstmt)
close(conn)
```

Insert Data Using SQL Prepared Statement

Create an SQL prepared statement to insert data from MATLAB® into a Microsoft® SQL Server® database using a JDBC database connection. Use the `INSERT` SQL statement for the SQL query. Execute the SQL prepared statement and display the results.

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MSSQLServerJDBCAuth';
conn = database(datasource, '', '');
```

Import data from the database using the `sqlread` function. Display the last few rows of data in the database table `inventoryTable`.

```
tablename = "inventoryTable";
data = sqlread(conn,tablename);
tail(data,3)
```

```
ans=3x4 table
    productNumber    Quantity    Price    inventoryDate
    _____    _____    _____    _____
         11             567         0    {'2012-09-11 00:30:24'}
         12            1278         0    {'2010-10-29 18:17:47'}
         13            1700        14.5    {'2009-05-24 10:58:59'}
```

Create an SQL prepared statement for inserting data using the JDBC database connection. The question marks in the `INSERT` SQL statement indicate it is an SQL prepared statement. This statement inserts data from MATLAB into the database table `inventoryTable`.

```
query = "INSERT INTO inventoryTable VALUES(?,?,?,?)";
pstmt = databasePreparedStatement(conn,query)
```

```
pstmt =
    SQLPreparedStatement with properties:
        SQLQuery: "INSERT INTO inventoryTable values(?,?,?,?)"
        ParameterCount: 4
        ParameterTypes: ["numeric"    "numeric"    "numeric"    "string"]
        ParameterValues: {[] [] [] []}
```

`pstmt` is an `SQLPreparedStatement` object with these properties:

- `SQLQuery` — SQL prepared statement query
- `ParameterCount` — Parameter count
- `ParameterTypes` — Parameter types
- `ParameterValues` — Parameter values

Bind parameter values in the SQL prepared statement. Select all parameters in the SQL prepared statement using their numeric indices. Specify the values to bind for the product number, quantity, price, and inventory date. Match the format of dates in the database. The `bindParamValues` function updates the values in the `ParameterValues` property of the `pstmt` object.

```
selection = [1 2 3 4];
values = {20,1000,55,"2019-04-25 00:00:00.000"};
pstmt = bindParamValues(pstmt,selection,values)
```

```
pstmt =
    SQLPreparedStatement with properties:
        SQLQuery: "INSERT INTO inventoryTable values(?,?,?,?)"
        ParameterCount: 4
        ParameterTypes: ["numeric"    "numeric"    "numeric"    "string"]
        ParameterValues: {[20] [1000] [55] ["2019-04-25 00:00:00.000"]}
```

Insert data from MATLAB into the database using the bound parameter values. Execute the SQL INSERT statement using the `execute` function.

```
execute(conn,pstmt)
```

Display the inserted data in the database table `inventoryTable`. The last row in the table contains the inserted data.

```
tablename = "inventoryTable";
data = sqlread(conn,tablename);
tail(data,4)
```

```
ans=4x4 table
   productNumber  Quantity  Price  inventoryDate
   _____  _____  _____  _____
           11           567         0  {'2012-09-11 00:30:24' }
           12          1278         0  {'2010-10-29 18:17:47' }
           13          1700        14.5  {'2009-05-24 10:58:59' }
           20          1000         55  {'2019-04-25 00:00:00.000' }
```

Close the SQL prepared statement and database connection.

```
close(pstmt)
close(conn)
```

Update Data Using SQL Prepared Statement

Create an SQL prepared statement to update data in a Microsoft® SQL Server® database using a JDBC database connection. Use the UPDATE SQL statement for the SQL query. Execute the SQL prepared statement and display the results.

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MSSQLServerJDBCAuth';
conn = database(datasource, '', '');
```

Import data from the database using the `sqlread` function. Display the first few rows of data in the database table `inventoryTable`.

```
tablename = "inventoryTable";
data = sqlread(conn,tablename);
head(data,3)
```

```
ans=3x4 table
   productNumber  Quantity  Price  inventoryDate
   _____  _____  _____  _____
           1          1700        14.5  {'2014-09-23 09:38:34' }
           2          1200         9  {'2014-07-08 22:50:45' }
           3           356         17  {'2014-05-14 07:14:28' }
```


Create an SQL prepared statement for updating data using the JDBC database connection. The question marks in the UPDATE SQL statement indicate it is an SQL prepared statement. This statement updates data in the database table `inventoryTable`.

```
query = strcat("UPDATE inventoryTable SET Quantity = ? ", ...
              "WHERE productNumber = ?");
pstmt = databasePreparedStatement(conn,query)
```

```
pstmt =
  SQLPreparedStatement with properties:
      SQLQuery: "UPDATE inventoryTable SET Quantity = ? WHERE productNumber = ?"
      ParameterCount: 2
      ParameterTypes: ["numeric"    "numeric"]
      ParameterValues: {[] []}
```

`pstmt` is an `SQLPreparedStatement` object with these properties:

- `SQLQuery` — SQL prepared statement query
- `ParameterCount` — Parameter count
- `ParameterTypes` — Parameter types
- `ParameterValues` — Parameter values

Bind parameter values in the SQL prepared statement. Select all parameters in the SQL prepared statement using their numeric indices. Specify the values to bind for the quantity and product number. The `bindParamValues` function updates the values in the `ParameterValues` property of the `pstmt` object.

```
selection = [1 2];
values = {2000,1};
pstmt = bindParamValues(pstmt,selection,values)
```

```
pstmt =
  SQLPreparedStatement with properties:
      SQLQuery: "UPDATE inventoryTable SET Quantity = ? WHERE productNumber = ?"
      ParameterCount: 2
      ParameterTypes: ["numeric"    "numeric"]
      ParameterValues: {[2000] [1]}
```

Update data in the database using the bound parameter values. Execute the SQL UPDATE statement using the `execute` function.

```
execute(conn,pstmt)
```

Display the updated data in the database table `inventoryTable`. The first row in the table contains the updated quantity.

```
tablename = "inventoryTable";
data = sqlread(conn,tablename);
head(data,3)
```

```
ans=3x4 table
  productNumber  Quantity  Price  inventoryDate
  _____  _____  _____  _____
```

```

1          2000      14.5    {'2014-09-23 09:38:34'}
2          1200       9      {'2014-07-08 22:50:45'}
3           356      17      {'2014-05-14 07:14:28'}

```

Close the SQL prepared statement and database connection.

```

close(pstmt)
close(conn)

```

Delete Data Using SQL Prepared Statement

Create an SQL prepared statement to delete data in a Microsoft® SQL Server® database using a JDBC database connection. Use the DELETE SQL statement for the SQL query. Execute the SQL prepared statement and display the results.

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```

datasource = 'MSSQLServerJDBCAuth';
conn = database(datasource, '', '');

```

Import data from the database using the `sqlread` function. Display rows of data in the database table `productTable`. The data contains rows for product numbers 16 through 20, which you will delete later in this example.

```

tablename = "productTable";
data = sqlread(conn, tablename)

```

```

data=15x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
  -----
      9          1.2597e+05          1003           13  {'Victorian Doll' }
      8          2.1257e+05          1001           5   {'Train Set'      }
      7          3.8912e+05          1007          16   {'Engine Kit'     }
      2          4.0031e+05          1002           9   {'Painting Set'   }
      4          4.0034e+05          1008          21   {'Space Cruiser'  }
      1          4.0035e+05          1001          14   {'Building Blocks'}
      5          4.0046e+05          1005           3   {'Tin Soldier'    }
      6          4.0088e+05          1004           8   {'Sail Boat'      }
      3           4.01e+05          1009          17   {'Slinky'         }
     10          8.8865e+05          1006          24   {'Teddy Bear'     }
     16          5.6789e+05          1001          10   {'Magnetic Links' }
     17           5.688e+05          1002          15   {'Hot Rod'        }
     18           5.679e+05          1003          20   {'Doll House'     }
     19          5.7761e+05          1004          25   {'Plush Monkey'   }
     20          5.0034e+05          1005          30   {'Kitchen Set'    }

```

Create an SQL prepared statement for deleting data using the JDBC database connection. The question marks in the DELETE SQL statement indicate it is an SQL prepared statement. This statement deletes data in the database table `productTable` for a specified range of product numbers.

```
query = strcat("DELETE FROM productTable ", ...
    "WHERE productNumber > ? AND productNumber < ?");
pstmt = databasePreparedStatement(conn,query)
```

```
pstmt =
    SQLPreparedStatement with properties:
        SQLQuery: "DELETE FROM productTable WHERE productNumber > ? AND productNumber < ?"
        ParameterCount: 2
        ParameterTypes: ["numeric"    "numeric"]
        ParameterValues: {[]  []}
```

pstmt is an SQLPreparedStatement object with these properties:

- SQLQuery — SQL prepared statement query
- ParameterCount — Parameter count
- ParameterTypes — Parameter types
- ParameterValues — Parameter values

Bind parameter values in the SQL prepared statement. Select all parameters in the SQL prepared statement using their numeric indices. Specify the values to bind for the range of product numbers between 15 and 21 (exclusive). The bindParamValues function updates the values in the ParameterValues property of the pstmt object.

```
selection = [1 2];
values = {15,21};
pstmt = bindParamValues(pstmt,selection,values)
```

```
pstmt =
    SQLPreparedStatement with properties:
        SQLQuery: "DELETE FROM productTable WHERE productNumber > ? AND productNumber < ?"
        ParameterCount: 2
        ParameterTypes: ["numeric"    "numeric"]
        ParameterValues: {[15]  [21]}
```

Delete data in the database using the bound parameter values. Execute the SQL DELETE statement using the execute function.

```
execute(conn,pstmt)
```

Display data in the database table productTable. The rows with product numbers 16 through 20 are no longer in the table.

```
tablename = "productTable";
data = sqlread(conn,tablename)
```

```
data=10x5 table
    productNumber    stockNumber    supplierNumber    unitCost    productDescription
    _____    _____    _____    _____    _____
         9         1.2597e+05         1003             13    {'Victorian Doll' }
         8         2.1257e+05         1001              5    {'Train Set'      }
         7         3.8912e+05         1007             16    {'Engine Kit'     }
         2         4.0031e+05         1002              9    {'Painting Set'   }
```

4	4.0034e+05	1008	21	{'Space Cruiser' }
1	4.0035e+05	1001	14	{'Building Blocks' }
5	4.0046e+05	1005	3	{'Tin Soldier' }
6	4.0088e+05	1004	8	{'Sail Boat' }
3	4.01e+05	1009	17	{'Slinky' }
10	8.8865e+05	1006	24	{'Teddy Bear' }

Close the SQL prepared statement and database connection.

```
close(pstmt)
close(conn)
```

Call Stored Procedure Using SQL Prepared Statement

Create an SQL prepared statement to call a stored procedure in a Microsoft® SQL Server® database using a JDBC database connection. Use the CALL SQL statement for the SQL query. Execute the SQL prepared statement and display the results.

For this example, the SQL Server database contains the stored procedure `getSupplierInfo`, which returns the information for suppliers in a specified city. This code defines the procedure.

```
CREATE PROCEDURE dbo.getSupplierInfo
    (@cityName varchar(20))
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result
    -- sets from interfering with SELECT statements.
    SET NOCOUNT ON
    SELECT * FROM dbo.suppliers WHERE City = @cityName
END
```

For SQL Server, the statement `SET NOCOUNT ON` suppresses the results of `INSERT`, `UPDATE`, and non-`SELECT` statements preceding the final `SELECT` query, so that you can import the results of the `SELECT` query.

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
conn = database(datasource, '', '');
```

Create an SQL prepared statement for calling the stored procedure using the JDBC database connection. The question marks in the CALL SQL statement indicate it is an SQL prepared statement. This statement calls the `getSupplierInfo` stored procedure in the database.

```
query = "{CALL dbo.getSupplierInfo(?)}";
pstmt = databasePreparedStatement(conn, query)

pstmt =
    SQLPreparedStatement with properties:
        SQLQuery: "{CALL dbo.getSupplierInfo(?)}"
        ParameterCount: 1
        ParameterTypes: "string"
```

```
ParameterValues: {}
```

`pstmt` is an `SQLPreparedStatement` object with these properties:

- `SQLQuery` — SQL prepared statement query
- `ParameterCount` — Parameter count
- `ParameterTypes` — Parameter types
- `ParameterValues` — Parameter values

Bind parameter values in the SQL prepared statement. Select the parameter in the SQL prepared statement using its numeric index. Specify the value to bind as the city `New York`. The `bindParamValues` function updates the values in the `ParameterValues` property of the `pstmt` object.

```
selection = [1];
values = "New York";
pstmt = bindParamValues(pstmt, selection, values)
```

```
pstmt =
  SQLPreparedStatement with properties:
      SQLQuery: "{CALL dbo.getSupplierInfo(?)}"
      ParameterCount: 1
      ParameterTypes: "string"
      ParameterValues: [{"New York"}]
```

Display the results of the stored procedure. Execute the SQL `CALL` statement using the `fetch` function. The SQL prepared statement returns all information for suppliers located in New York City.

```
results = fetch(conn, pstmt)
```

```
results=2x5 table
  SupplierNumber      SupplierName      City      Country      FaxNumber
-----
      1001      {'Wonder Products' }      {'New York'}      {'United States'}      {'212 435 161
      1006      {'ACME Toy Company'}      {'New York'}      {'United States'}      {'212 435 161
```

Close the SQL prepared statement and database connection.

```
close(pstmt)
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with the database function.

Note The `databasePreparedStatement` function supports a JDBC database connection only.

query — SQL prepared statement

character vector | string scalar

SQL prepared statement query, specified as a character vector or string scalar that contains one of these SQL statements:

- SELECT
- INSERT
- UPDATE
- DELETE
- CALL

Example: "SELECT * FROM inventoryTable WHERE inventoryDate > ? AND inventoryDate < ?" selects all data from the database table `inventoryTable` with an inventory date between two parameters.

Example: "INSERT INTO inventoryTable VALUES(?, ?, ?, ?)" inserts data into the database table `inventoryTable` based on parameters for four database columns.

Data Types: `char` | `string`

Output Arguments**pstmt — SQL prepared statement**

SQLPreparedStatement object

SQL prepared statement, returned as an `SQLPreparedStatement` object.

Version History**Introduced in R2019b****See Also**`database` | `close` | `bindParamValues` | `close` | `fetch` | `execute`**Topics**

"Import Data Using SQL Prepared Statement with Multiple Parameter Values" on page 5-75

"SQL Prepared Statement Error Messages" on page 3-15

bindParamValues

Package: database.preparedstatement

Bind values to parameters

Syntax

```
pstmt = bindParamValues(pstmt,selection,values)
```

Description

`pstmt = bindParamValues(pstmt,selection,values)` binds parameters specified in the `selection` argument to the values specified in the `values` argument for an `SQLPreparedStatement` object.

Examples

Import Data Using SQL Prepared Statement

Create an SQL prepared statement to import data from a Microsoft® SQL Server® database using a JDBC database connection. Use the SELECT SQL statement for the SQL query. Import the data from the database and display the results.

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MSSQLServerJDBCAuth';
conn = database(datasource, '', '');
```

Create an SQL prepared statement for importing data from the SQL Server database using the JDBC database connection. The question marks in the SELECT SQL statement indicate it is an SQL prepared statement. This statement selects all data from the database table `inventoryTable` for the inventory that has an inventory date within a specified date range.

```
query = strcat("SELECT * FROM inventoryTable ", ...
  "WHERE inventoryDate > ? AND inventoryDate < ?");
pstmt = databasePreparedStatement(conn,query)
```

```
pstmt =
  SQLPreparedStatement with properties:
```

```
    SQLQuery: "SELECT * FROM inventoryTable where inventoryDate > ? AND inventoryDate < ?"
    ParameterCount: 2
    ParameterTypes: ["string"    "string"]
    ParameterValues: {[] []}
```

`pstmt` is an `SQLPreparedStatement` object with these properties:

- `SQLQuery` — SQL prepared statement query

- `ParameterCount` — Parameter count
- `ParameterTypes` — Parameter types
- `ParameterValues` — Parameter values

Bind parameter values in the SQL prepared statement. Select both parameters in the SQL prepared statement using their numeric indices. Specify the values to bind as the inventory date range between January 1, 2014, and December 31, 2014. Match the format of dates in the database. The `bindParamValues` function updates the values in the `ParameterValues` property of the `pstmt` object.

```
selection = [1 2];
values = {"2014-01-01 00:00:00.000", ...
         "2014-12-31 00:00:00.000"};
pstmt = bindParamValues(pstmt,selection,values)
```

```
pstmt =
  SQLPreparedStatement with properties:
```

```
    SQLQuery: "SELECT * FROM inventoryTable where inventoryDate > ? AND inventoryDate < ?"
  ParameterCount: 2
  ParameterTypes: ["string" "string"]
  ParameterValues: [{"2014-01-01 00:00:00.000"} {"2014-12-31 00:00:00.000"}]
```

Import data from the database using the `fetch` function and bound parameter values. The results contain four rows of data that represent all inventory with an inventory date between January 1, 2014 and December 31, 2014.

```
results = fetch(conn,pstmt)
```

```
results=4x4 table
  productNumber  Quantity  Price  inventoryDate
  -----
           1         1700    14.5  {'2014-09-23 09:38:34'}
           2         1200     9     {'2014-07-08 22:50:45'}
           3          356    17     {'2014-05-14 07:14:28'}
           7         6034    16     {'2014-08-06 08:38:00'}
```

Close the SQL prepared statement and database connection.

```
close(pstmt)
close(conn)
```

Input Arguments

pstmt — SQL prepared statement

SQLPreparedStatement object

SQL prepared statement, specified as an SQLPreparedStatement object.

selection — Selected parameters

numeric scalar | numeric array

Selected parameters, specified as a numeric scalar for one index or a numeric array for multiple indices.

Example: 1

Example: [1 2 3]

Data Types: double

values — Parameter values

numeric scalar | string scalar | character vector | ...

Parameter values to bind for the selected parameters, specified as a numeric scalar, string scalar, character vector, `datetime` array, `logical`, or cell array. The function treats missing values as `NaN` or `missing`. Use a cell array to specify multiple values.

The values for each parameter must have one of the types specified by the `ParameterTypes` property of the `SQLPreparedStatement` object.

Example: `true`

Example: `"USA"`

Example: `{true, "USA", 2, datetime('now')}`

Data Types: double | logical | char | string | cell | datetime

Output Arguments

pstmt — SQL prepared statement

`SQLPreparedStatement` object

SQL prepared statement, returned as a `SQLPreparedStatement` object.

Version History

Introduced in R2019b

See Also

`database` | `close` | `databasePreparedStatement` | `close` | `fetch`

Topics

“Import Data Using SQL Prepared Statement with Multiple Parameter Values” on page 5-75

“SQL Prepared Statement Error Messages” on page 3-15

close

Package: `database.preparedstatement`

Close SQL prepared statement

Syntax

```
close(pstmt)
```

Description

`close(pstmt)` closes the SQL prepared statement specified by an `SQLPreparedStatement` object.

Examples

Import Data Using SQL Prepared Statement

Create an SQL prepared statement to import data from a Microsoft® SQL Server® database using a JDBC database connection. Use the `SELECT` SQL statement for the SQL query. Import the data from the database and display the results.

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MSSQLServerJDBCAuth';  
conn = database(datasource, '', '');
```

Create an SQL prepared statement for importing data from the SQL Server database using the JDBC database connection. The question marks in the `SELECT` SQL statement indicate it is an SQL prepared statement. This statement selects all data from the database table `inventoryTable` for the inventory that has an inventory date within a specified date range.

```
query = strcat("SELECT * FROM inventoryTable ", ...  
              "WHERE inventoryDate > ? AND inventoryDate < ?");  
pstmt = databasePreparedStatement(conn, query)
```

```
pstmt =  
  SQLPreparedStatement with properties:
```

```
      SQLQuery: "SELECT * FROM inventoryTable where inventoryDate > ? AND inventoryDate < ?"  
      ParameterCount: 2  
      ParameterTypes: ["string" "string"]  
      ParameterValues: {[] []}
```

`pstmt` is an `SQLPreparedStatement` object with these properties:

- `SQLQuery` — SQL prepared statement query
- `ParameterCount` — Parameter count

- `ParameterTypes` — Parameter types
- `ParameterValues` — Parameter values

Bind parameter values in the SQL prepared statement. Select both parameters in the SQL prepared statement using their numeric indices. Specify the values to bind as the inventory date range between January 1, 2014, and December 31, 2014. Match the format of dates in the database. The `bindParamValues` function updates the values in the `ParameterValues` property of the `pstmt` object.

```
selection = [1 2];
values = {"2014-01-01 00:00:00.000", ...
         "2014-12-31 00:00:00.000"};
pstmt = bindParamValues(pstmt,selection,values)

pstmt =
  SQLPreparedStatement with properties:
    SQLQuery: "SELECT * FROM inventoryTable where inventoryDate > ? AND inventoryDate < ?"
    ParameterCount: 2
    ParameterTypes: ["string" "string"]
    ParameterValues: [{"2014-01-01 00:00:00.000"} ["2014-12-31 00:00:00.000"]}
```

Import data from the database using the `fetch` function and bound parameter values. The results contain four rows of data that represent all inventory with an inventory date between January 1, 2014 and December 31, 2014.

```
results = fetch(conn,pstmt)
```

```
results=4x4 table
  productNumber  Quantity  Price  inventoryDate
  -----
           1         1700   14.5  {'2014-09-23 09:38:34'}
           2         1200    9     {'2014-07-08 22:50:45'}
           3          356   17     {'2014-05-14 07:14:28'}
           7         6034   16     {'2014-08-06 08:38:00'}
```

Close the SQL prepared statement and database connection.

```
close(pstmt)
close(conn)
```

Input Arguments

`pstmt` — SQL prepared statement

SQLPreparedStatement object

SQL prepared statement, specified as an SQLPreparedStatement object.

Version History

Introduced in R2019b

See Also

database | close | databasePreparedStatement | bindParamValues | fetch

Topics

“Import Data Using SQL Prepared Statement with Multiple Parameter Values” on page 5-75

“SQL Prepared Statement Error Messages” on page 3-15

datainsert

Package: database.odbcc

(To be removed) Export MATLAB data into database table

Note The `datainsert` function will be removed in a future release. Use the `sqlwrite` function instead. For details, see “Compatibility Considerations”.

Syntax

```
datainsert(conn,tablename,colnames,data)
```

Description

`datainsert(conn,tablename,colnames,data)` exports data from the MATLAB workspace and inserts it into existing columns of a database table using the database connection `conn`.

Examples

Export MATLAB Cell Array Data

Use an ODBC connection and a cell array to export inventory data from MATLAB into a MySQL database table.

Create a database connection `conn` to the MySQL database using the native ODBC interface. Here, this code assumes that you are connecting to an ODBC data source named `MySQL` with a user name and password. This database contains the table `inventoryTable` with these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

```
conn = database('MySQL','username','pwd');
```

Display the last rows in `inventoryTable` before inserting data.

```
curs = exec(conn,'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```

...
[14]    [2000]    [19.1000]    '2014-10-22 10:52...'
[15]    [1200]    [20.3000]    '2014-10-22 10:52...'
[16]    [1400]    [34.3000]    '1999-12-31 00:00...'

```

Create a cell array of column names for the database table `inventoryTable`.

```
colnames = {'productNumber','Quantity','Price','inventoryDate'};
```

Define a cell array of input data to insert.

```
data = {50 100 15.50 datestr(now,'yyyy-mm-dd HH:MM:SS')};
```

Insert the input data into the table `inventoryTable` using the database connection.

```
tablename = 'inventoryTable';
```

```
datainsert(conn,tablename,colnames,data)
```

Display the inserted data in `inventoryTable`.

```
curs = exec(conn,'SELECT * FROM inventoryTable');
```

```
curs = fetch(curs);
```

```
curs.Data
```

```
ans =
```

```

...
[15]    [1200]    [20.3000]    '2014-10-22 10:52...'
[16]    [1400]    [34.3000]    '1999-12-31 00:00...'
[50]     [ 100]    [15.5000]    '2014-10-22 11:29...'

```

The last row contains the inserted data.

After you finish working with the `cursor` object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

Export MATLAB Table Data

Use a JDBC connection and a MATLAB table to export inventory data from MATLAB into a MySQL database table.

Create a database connection `conn` to the MySQL database using the JDBC driver. Use the Vendor name-value pair argument of the `database` function to specify a connection to a MySQL database. Here, this code assumes that you are connecting to a database named `dbname` on a database server named `sname` with a user name and password. This database contains the table `inventoryTable` with these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

```
conn = database('dbname','username','pwd', ...
    'Vendor','MySQL', ...
    'Server','sname');
```

Display the last rows in `inventoryTable` before inserting data.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data

ans =

...
[14]    [2000]    [19.1000]    '2014-10-22 10:52...'
[15]    [1200]    [20.3000]    '2014-10-22 10:52...'
[16]    [1400]    [34.3000]    '1999-12-31 00:00...'
```

Create a cell array of column names for the database table `inventoryTable`.

```
colnames = {'productNumber', 'Quantity', 'Price', 'inventoryDate'};
```

Define the input data as a table.

```
data = table(50,100,15.50,{datestr(now, 'yyyy-mm-dd HH:MM:SS')}, ...
    'VariableNames', colnames);
```

Insert the input data into the table `inventoryTable` using the database connection.

```
tablename = 'inventoryTable';
datainsert(conn, tablename, colnames, data)
```

Display the inserted data in `inventoryTable`.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data

ans =

...
[15]    [1200]    [20.3000]    '2014-10-22 10:52...'
[16]    [1400]    [34.3000]    '1999-12-31 00:00...'
[50]    [ 100]    [15.5000]    '2014-10-22 11:29...'
```

The last row contains the inserted data.

After you finish working with the cursor object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

Export MATLAB Structure Data

Use an ODBC connection and a MATLAB structure to export inventory data from MATLAB into a MySQL database table.

Create a database connection `conn` to the MySQL database using the native ODBC interface. Here, this code assumes that you are connecting to an ODBC data source named `MySQL` with a user name and password. This database contains the table `inventoryTable` with these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

```
conn = database('MySQL','username','pwd');
```

Display the last rows in `inventoryTable` before inserting data.

```
curs = exec(conn,'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
...
[14]    [2000]    [19.1000]    '2014-10-22 10:52...'
[15]    [1200]    [20.3000]    '2014-10-22 10:52...'
[16]    [1400]    [34.3000]    '1999-12-31 00:00...'
```

Create a cell array of column names for the database table `inventoryTable`.

```
colnames = {'productNumber','Quantity','Price','inventoryDate'};
```

Define the input data as a structure.

```
data = struct('productNumber',50,'Quantity',100,'Price',15.50, ...
    'inventoryDate',datestr(now,'yyyy-mm-dd HH:MM:SS'));
```

Insert the input data into the table `inventoryTable` using the database connection.

```
tablename = 'inventoryTable';
datainsert(conn,tablename,colnames,data)
```

Display the inserted data in `inventoryTable`.

```
curs = exec(conn,'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
...
[15]    [1200]    [20.3000]    '2014-10-22 10:52...'
[16]    [1400]    [34.3000]    '1999-12-31 00:00...'
[50]    [ 100]    [15.5000]    '2014-10-22 11:29...'
```

The last row contains the inserted data.

After you finish working with the cursor object, close it.

```
close(curs)
```


Close the database connection.

```
close(conn)
```

Export MATLAB Numeric Matrix Data

Use a JDBC connection and a numeric matrix to export sales data from MATLAB into a MySQL database table.

Create a database connection `conn` to the MySQL database using the JDBC driver. Use the Vendor name-value pair argument of `database` to specify a connection to a MySQL database. Here, this code assumes that you are connecting to a database named `dbname` on a database server named `sname` with a user name and password. This database contains the table `salesVolume` with the column `stockNumber` and columns for each month of the year.

```
conn = database('dbname','username','pwd', ...
    'Vendor','MySQL', ...
    'Server','sname');
```

Display the last rows in `salesVolume` before inserting data.

```
curs = exec(conn,'SELECT * FROM salesVolume');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
Columns 1 through 8
```

```
...
[470816] [3100] [9400] [1540] [1500] [1350] [1190] [ 900]
[510099] [ 235] [1800] [1040] [ 900] [ 750] [ 700] [ 400]
[899752] [ 123] [1700] [ 823] [ 701] [ 689] [ 621] [ 545]
```

```
Columns 9 through 13
```

```
...
[867] [ 923] [1400] [ 3000] [35000]
[350] [ 500] [ 100] [ 3000] [18000]
[421] [ 495] [ 650] [ 4200] [11000]
```

Create a cell array of column names for the database table `salesVolume`.

```
colnames = {'stockNumber','January','February' ...
    'March','April','May', ...
    'June','July','August', ...
    'September','October','November', ...
    'December'};
```

Define the numeric matrix `data` that contains the sales volume data.

```
data = [777666,0,350,400,450,250,450,500,515, ...
    235,100,300,600];
```

Insert the contents of `data` into the table `salesVolume` using the database connection.

```
tablename = 'salesVolume';
datainsert(conn,tablename,colnames,data)
```

Display the inserted data in `salesVolume`.

```
curs = exec(conn, 'SELECT * FROM salesVolume');
curs = fetch(curs);
curs.Data

ans =

Columns 1 through 8

...
[510099] [ 235] [1800] [1040] [ 900] [ 750] [ 700] [ 400]
[899752] [ 123] [1700] [ 823] [ 701] [ 689] [ 621] [ 545]
[777666] [  0] [ 350] [ 400] [ 450] [ 250] [ 450] [ 500]

Columns 9 through 13

...
[350] [ 500] [ 100] [ 3000] [18000]
[421] [ 495] [ 650] [ 4200] [11000]
[515] [ 235] [ 100] [ 300] [ 600]
```

The last row contains the inserted data.

After you finish working with the cursor object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the database function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

colnames — Database table column names

cell array of character vectors | string array

Database table column names, specified as a cell array of one or more character vectors or a string array to denote the columns in the existing database table `tablename`.

Example: {'col1', 'col2', 'col3'}

Data Types: cell | string

data — Insert data

cell array | numeric matrix | table | structure | dataset

Insert data, specified as a cell array, numeric matrix, table, structure, or dataset array.

If you are connecting to a database using a JDBC driver, then convert the insert data to a supported format before running `datainsert`. If `data` contains MATLAB dates, times, or timestamps, use this formatting:

- Dates must be character vectors of the form `yyyy-mm-dd`.
- Times must be character vectors of the form `HH:MM:SS`.
- Timestamps must be character vectors of the form `yyyy-mm-dd HH:MM:SS.FFF`.

The database preference settings `NullNumberWrite` and `NullStringWrite` do not apply to this function. If `data` contains `null` entries and NaNs, convert these entries to an empty value `''`.

The `datainsert` function supports inserting MATLAB date numbers and NaNs when `data` is a numeric matrix. Date numbers inserted into database date and time columns convert to `java.sql.Date`. Upon insertion into the target database, any converted date and time data accurately reverts to the native database format.

If `data` is a structure, then field names in the structure must match `colnames`.

If `data` is a table or a dataset array, then the variable names in the table or dataset array must match `colnames`.

Tips

- When you establish a database connection using a JDBC driver, `datainsert` performs faster than `fastinsert`.
- `datainsert` uses the SQL `TRANSACTION` statement to insert records with faster performance for these databases:
 - Microsoft SQL Server
 - MySQL
 - Oracle
 - PostgreSQL

For other databases, refer to your database documentation to start a transaction manually. Before running `datainsert`, use `exec` to start the transaction.

- The value of the `AutoCommit` property in the `connection` object determines whether `datainsert` automatically commits the data to the database.
 - To view the `AutoCommit` value, access it using the `connection` object; for example, `conn.AutoCommit`.
 - To set the `AutoCommit` value, use the corresponding name-value pair argument in the `database` function.
 - To commit the data to the database, use the `commit` function or issue an SQL `COMMIT` statement using the `exec` function.
 - To roll back the data, use `rollback` or issue an SQL `ROLLBACK` statement using the `exec` function.

Alternative Functionality

To export MATLAB data into a database, you can use the `fastinsert` and `insert` functions. For maximum performance, use `datainsert`.

Version History

Introduced in R2011a

R2018a: `datainsert` function will be removed

Not recommended starting in R2018a

The `datainsert` function will be removed in a future release. Use the `sqlwrite` function instead. Some differences between the workflows require updates to your code.

Update Code

In prior releases, you exported data from the MATLAB workspace into a database by using the `datainsert` function and four input arguments. For example:

```
colnames = {'productNumber', 'Quantity', 'Price', 'inventoryDate'};
data = table(50,100,15.50,{datestr(now,'yyyy-mm-dd HH:MM:SS')}, ...
    'VariableNames',colnames);
tablename = 'inventoryTable';
datainsert(conn,tablename,colnames,data)
```

Now the `sqlwrite` function requires only three input arguments.

```
colnames = {'productNumber', 'Quantity', 'Price', 'inventoryDate'};
data = table(50,100,15.50,{datestr(now,'yyyy-mm-dd HH:MM:SS')}, ...
    'VariableNames',colnames);
tablename = 'inventoryTable';
sqlwrite(conn,tablename,data)
```

See Also

`sqlwrite` | `database` | `fastinsert` | `insert` | `select` | `close`

Topics

“Insert Data into Database Table” on page 5-61

“Export Data Using Bulk Insert” on page 5-13

“Replace Existing Data in Database” on page 5-12

“Roll Back Data After Updating Record” on page 5-9

“Data Type Support” on page 1-3

External Websites

SQL Tutorial

Database Explorer

Configure, explore, and import database data

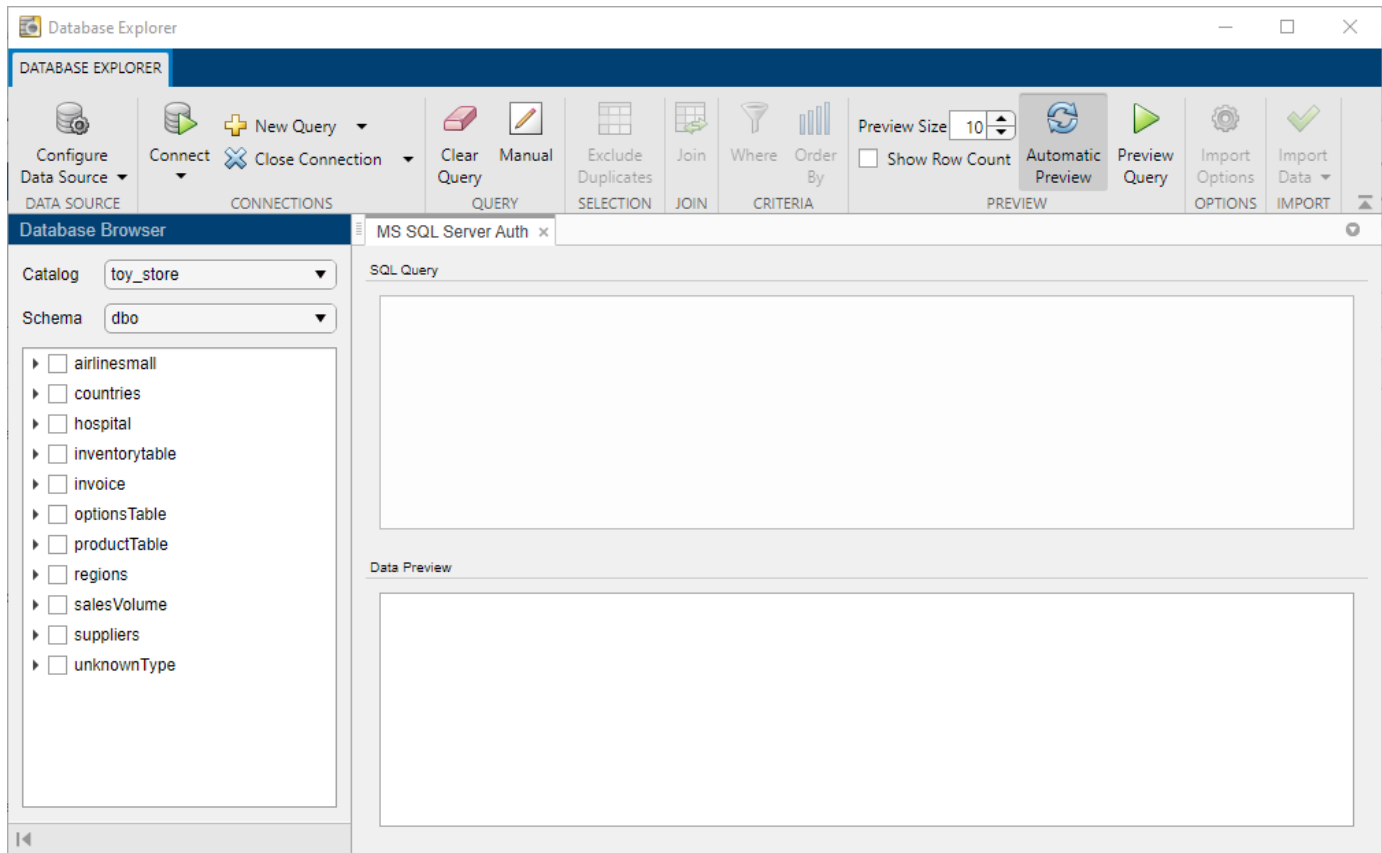
Description

The **Database Explorer** app lets you quickly connect to a database, explore the database data, and import data into the MATLAB workspace in a visual way. If you have minimal proficiency writing SQL queries or want to browse the data in your database quickly, use this app to interact with your database.

Using the Database Explorer app, you can:

- Create and configure ODBC and JDBC data sources.
- Establish multiple connections to the same or different databases.
- Select tables and columns of interest.
- Fine-tune selections using SQL query criteria.
- Preview selected data.
- Customize import options.
- Import selected data into the MATLAB workspace for analysis.
- Save generated SQL queries.
- Generate MATLAB code.

To watch an introductory video, see [Using the Database Explorer App](#).



Open the Database Explorer App

- MATLAB Toolstrip: On the **Apps** tab, click the **Show more** arrow to open the apps gallery. Then, under **Database Connectivity and Reporting**, click **Database Explorer**.
- MATLAB command prompt: Enter databaseExplorer.

Examples

Preview Rows in Single Table

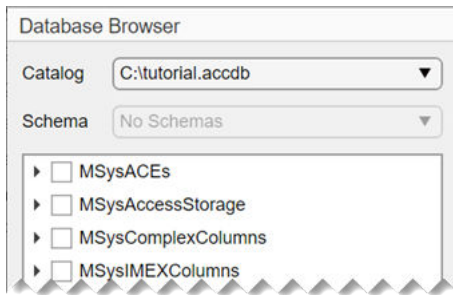
Connect to a Microsoft Access database using the Database Explorer app. Then, select columns from a single table and preview the data. The app previews query results by default.

Set up the data source for the `tutorial.accdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

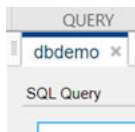
In the **Connections** section of the **Database Explorer** tab, click **Connect** and select the data source for the connection. The connection dialog box opens. Leave the user name and password blank, and click **Connect**.

Note For other databases, the Catalog and Schema dialog box opens. Select the name of the catalog and schema from the **Catalog** and **Schema** lists, as appropriate for your database.

The Database Explorer app creates a connection to the Microsoft Access database. The **Database Browser** pane displays the available tables in the database.



The data source tab, which is named **dbdemo**, appears to the right of the **Database Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.



For any table, you can select the table information in these ways:

- To select tables, click the database table name in the **Database Browser** pane. The Database Explorer app updates the **SQL Query** pane with an SQL query that selects all columns and rows from the chosen table. Simultaneously, the Database Explorer app updates the **Data Preview** pane with a preview of the query results. The first 10 rows of data appear in the **Data Preview** pane by default.
- To select individual columns from a selected table, expand the table name node in the **Database Browser** tree view. Select specific check boxes to choose individual table columns and display them in the **Data Preview** pane. The SQL query adjusts to each selection automatically.

Note The order of the columns in the **Data Preview** pane matches the order in which you select them in the **Database Browser** pane.

Select the table name **inventorytable**.

To change the data you see, select or clear check boxes in the **Database Browser** pane. The app updates the SQL query in the **SQL Query** pane. The app updates the data in the **Data Preview** pane.

The **Data Preview** pane displays 10 rows. The total number of rows selected in the database appears, within parentheses, next to the name of the pane, **Data Preview**. Change the number of rows by selecting or entering a value in the **Preview Size** box in the **Preview** section of the **Database Explorer** tab. Select the value 20. The number of rows adjusts in the **Data Preview** pane.

Note The value in the **Preview Size** box controls the maximum number of rows displayed in the **Data Preview** pane. If this value is larger than the total number of rows in the query results, then the total number of rows is displayed, within parentheses, next to the name of the pane, **Data Preview**.

The screenshot shows the Microsoft Access Database Explorer interface. The top toolbar includes options for 'Configure Data Source', 'Connect', 'Close Connection', 'Clear Query', 'Manual Query', 'Exclude Duplicates', 'Join', 'Where', 'Order By', 'Preview Size', 'Show Row Count', 'Automatic Preview', 'Preview Query', 'Import Options', and 'Import Data'. The 'Database Browser' pane on the left shows a tree view of the database 'C:\tutorial.accdb' with various system tables and user tables. The 'inventorytable' is selected. The 'SQL Query' pane contains the query: `SELECT * FROM inventorytable`. The 'Data Preview (All 13 Rows)' pane shows the following data:

productnumber	quantity	price	inventorydate
1	1700	14.5000	2014-09-23 09:38:34
2	1200	9	2014-07-08 22:50:45
3	356	17	2014-05-14 07:14:28
4	2580	21	2013-06-08 14:24:33
5	9000	3	2012-09-14 15:00:25
6	4540	8	2013-12-25 19:45:00
7	6034	16	2014-08-06 08:38:00
8	8350	5	2011-06-18 11:45:35
9	2339	13	2011-02-09 12:50:59
10	723	24	2012-03-14 13:13:09
11	567	0	2012-09-11 00:30:24

You can sort the rows of data by a specific column. In the **Criteria** section, click **Order By**. The **Order By** tab is displayed in the toolbar.

In the **Add** section, in the **Column** list, select the column `price`. In the **Add** section, click **Add Sort**. The Database Explorer app sorts the data in ascending order in the **Data Preview** pane. To change the order, click **Descending** in the **Edit** section.

The screenshot shows the Database Explorer interface. The 'ORDER BY' tab is active, displaying a list of columns with 'price' selected and 'DESC' sorting. The 'SQL Query' window contains the following query:

```
SELECT *
FROM inventorytable
ORDER BY price DESC
```

The 'Data Preview (All 13 Rows)' window shows the following table:

productnumber	quantity	price	inventorydate
10		723	24 2012-03-14 13:13:09
4		2580	21 2013-06-08 14:24:33
3		356	17 2014-05-14 07:14:28
7		6034	16 2014-08-06 08:38:00
13		1700	14.5000 2009-05-24 10:58:59
1		1700	14.5000 2014-09-23 09:38:34
9		2339	13 2011-02-09 12:50:59
2		1200	9 2014-07-08 22:50:45
6		4540	8 2013-12-25 19:45:00
8		8350	5 2011-06-18 11:45:35
5		9000	3 2012-09-14 15:00:25

Note To add more sorts, select another column from the **Column** list and click **Add Sort**. You can change the position of the sort in the SQL query by clicking it in the list in the **Edit** section, and then clicking **Move Up** or **Move Down**.

In the **Close** section, click **Close Order By** to close the **Order By** tab.

In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

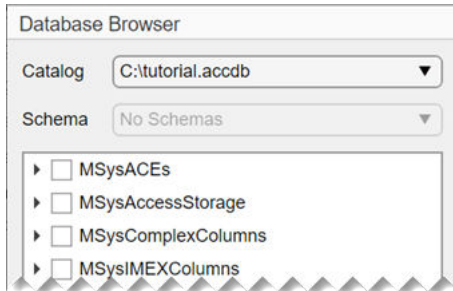
Join Multiple Tables and Import Query Results

Connect to a Microsoft Access database using the Database Explorer app. Then, join data in multiple tables by selecting columns in the tables. The app previews query results by default. After previewing the data, import all query results into the MATLAB Workspace and perform simple data analysis.

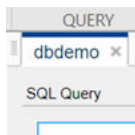
Set up the data source for the `tutorial.accdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

In the **Connections** section of the **Database Explorer** tab, click **Connect** and select the data source for the connection. The connection dialog box opens. Leave the user name and password blank, and click **Connect**.

The Database Explorer app creates a connection to the Microsoft Access database. The **Database Browser** pane displays the available tables in the database.



The data source tab, which is named `dbdemo`, appears to the right of the **Database Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.



In the **Database Browser** pane, select the `inventorytable` table as the first table for the join. The Database Explorer app updates the **SQL Query** pane with an SQL query that selects all columns and rows from the `inventorytable` table. Simultaneously, the app updates the **Data Preview** pane with a preview of the query results. The first 10 rows of data appear in the pane by default.

The screenshot shows the Database Explorer interface. The left pane shows the 'Database Browser' with the 'inventorytable' selected. The central pane shows the SQL Query window with the following query:

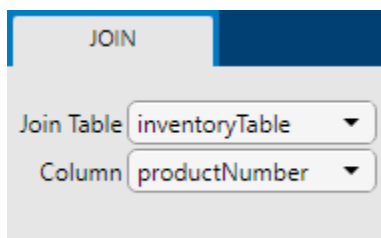
```
SELECT *
FROM inventorytable
```

Below the query window, the 'Data Preview (First 10 Rows)' table is displayed:

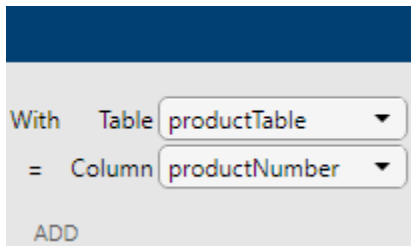
productnumber	quantity	price	inventorydate
1	1700	14.5000	2014-09-23 09:38:34
2	1200	9	2014-07-08 22:50:45
3	356	17	2014-05-14 07:14:28
4	2580	21	2013-06-08 14:24:33
5	9000	3	2012-09-14 15:00:25
6	4540	8	2013-12-25 19:45:00
7	6034	16	2014-08-06 08:38:00
8	8350	5	2011-06-18 11:45:35
9	2339	13	2011-02-09 12:50:59
10	723	24	2012-03-14 13:13:09

In the **Join** section, click **Join** to display the **Join** tab in the toolbar. In the **Add** section, the name of the table selected in the **Database Browser** pane appears in the left **Table** list. For details about joining tables, see “Join Tables Using Database Explorer App” on page 4-10.

In the left **Column** list, select the name of the shared column `productnumber`.



In the right **Table** list, select the table `producttable` as the table to join. Select the name of the shared column `productnumber` in this table in the right **Column** list.



In the **Add** section, click **Add Join**. The **Join Diagram** pane displays a pictorial representation of the join between the selected tables. The **SQL Query** pane updates the SQL query with the new join. The **Data Preview** pane reflects the results of the updated SQL query.

The Database Explorer app selects the inner join by default.

Note Some databases do not support all join types.

The screenshot shows the Database Explorer interface with the following components:

- JOIN Pane:** Shows 'Join Table' as 'inventorytable' and 'With Table' as 'producttable'. The 'Column' is 'productnumber'. The join type is 'INNER JOIN'. Buttons for 'Remove Join', 'Inner', 'Full', 'Left', 'Right', 'Close Join', and 'CLOSE' are visible.
- Database Browser:** Shows a tree view of the database structure. 'inventorytable' is selected under the 'producttable' schema.
- SQL Query:** Displays the following query:


```
SELECT inventorytable.productnumber,
        inventorytable.quantity,
        inventorytable.price,
        inventorytable.inventorydate
FROM ( inventorytable
INNER JOIN producttable
ON inventorytable.productnumber = producttable.productnumber)
```
- Data Preview (First 10 Rows):**

productnumber	quantity	price	inventorydate
9	2339	13	2011-02-09 12:50:59
8	8350	5	2011-06-18 11:45:35
7	6034	16	2014-08-06 08:38:00
2	1200	9	2014-07-08 22:50:45
4	2580	21	2013-06-08 14:24:33
1	1700	14.5000	2014-09-23 09:38:34
5	9000	3	2012-09-14 15:00:25
6	4540	8	2013-12-25 19:45:00
3	356	17	2014-05-14 07:14:28
10	723	24	2012-03-14 13:13:09
- Join Diagram:** A visual representation showing 'inventorytable' and 'producttable' connected by an 'INNER JOIN' relationship.

In the **Close** section, click **Close Join** to close the **Join** tab.

In the tree view of the **Database Browser** pane, select **productdescription** under **producttable**. The **SQL Query** and **Data Preview** panes update with the selected table column.

Add filter criteria to the SQL query. In the **Criteria** section, click **Where** to display the **Where** tab in the toolstrip.

Filter the SQL query results for prices greater than \$10. In the **Add** section, in the **Column** list, select `inventorytable.price`. Select the `>` operator for the filter in the **Operator** list. Enter `10` in the **Value** list. Click **Add Filter**.

Note If you enter filters using the `LIKE` or `NOT LIKE` operators, then enter the value in single quotes to represent a string.

The **SQL Query** and **Data Preview** panes display the updated query results based on the new filter with the `WHERE` condition.

The screenshot shows the Database Explorer interface with the following components:


- WHERE Filter Configuration:** Column: `inventorytable.price`, Operator: `>`, Value: `10`. Buttons: Add Filter, Update Filter, Remove Filter, And, Or, Move Up, Move Down, Close Where, CLOSE.
- Database Browser:** Catalog: `C:\tutorial.accdb`, Schema: `No Schemas`. A tree view shows various system tables and user tables, with `producttable` selected and `productdescription` checked.
- SQL Query:**

```
SELECT inventorytable.productnumber,
inventorytable.quantity,
inventorytable.price,
inventorytable.inventorydate,
producttable.productdescription
FROM ( inventorytable
INNER JOIN producttable
ON inventorytable.productnumber = producttable.productnumber)
WHERE inventorytable.price > 10
```
- Data Preview (All 6 Rows):**

productnumber	quantity	price	inventorydate	productdescription
1	1700	14.5000	2014-09-23 09:38:34	Building Blocks
3	356	17	2014-05-14 07:14:28	Slinky
4	2580	21	2013-06-08 14:24:33	Space Cruiser
7	6034	16	2014-08-06 08:38:00	Engine Kit
9	2339	13	2011-02-09 12:50:59	Victorian Doll
10	723	24	2012-03-14 13:13:09	Teddy Bear

In the **Close** section, click **Close Where** to close the **Where** tab.



Import all SQL query results into the MATLAB Workspace. In the **Import** section, click . In the Import Data dialog box, enter the name `data` for the MATLAB Workspace variable, and click **OK**. The MATLAB Workspace displays the table `data`.

Display the SQL query results at the command line.

```
data
```

```
data =
```

```
6x5 table
```

productnumber	quantity	price	inventorydate	productdescription
1	1700	14.5	'2014-09-23 09:38:34'	'Building Blocks'
3	356	17	'2014-05-14 07:14:28'	'Slinky'
4	2580	21	'2013-06-08 14:24:33'	'Space Cruiser'
...				

Find the maximum product price.

```
max(data.price)
```

```
ans =
```

```
24
```

In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

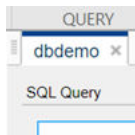
Join Tables Using Left Join and Import Query Results

Connect to a Microsoft Access database using the Database Explorer app. Then, create an SQL query that joins two tables using a left join. The Database Explorer app previews query results by default. After previewing the data, import all query results into the MATLAB workspace and perform simple data analysis.

Set up the data source for the `tutorial.accdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

In the **Connections** section of the **Database Explorer** tab, from the **Connect** list, select the data source for connection. The connection dialog box opens. Leave the user name and password blank, and click **Connect**.

The Database Explorer app creates a connection to the Microsoft Access database. The **Database Browser** pane displays the available tables in the database. The data source tab, which is named **dbdemo**, appears to the right of the **Database Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.



In the **Database Browser** pane, select the **suppliers** table as the first table for the join.

The Database Explorer app updates the **SQL Query** pane with an SQL query that selects all columns and rows from the **suppliers** table. Simultaneously, the Database Explorer app updates the **Data Preview** pane with a preview of the query results. The first 10 rows of data appear in the pane by default.

In the **Join** section, click **Join** to display the **Join** tab in the toolbar. In the **Add** section, the name of the table selected in the **Database Browser** pane appears in the left **Table** list. For details about joining tables, see “Join Tables Using Database Explorer App” on page 4-10.

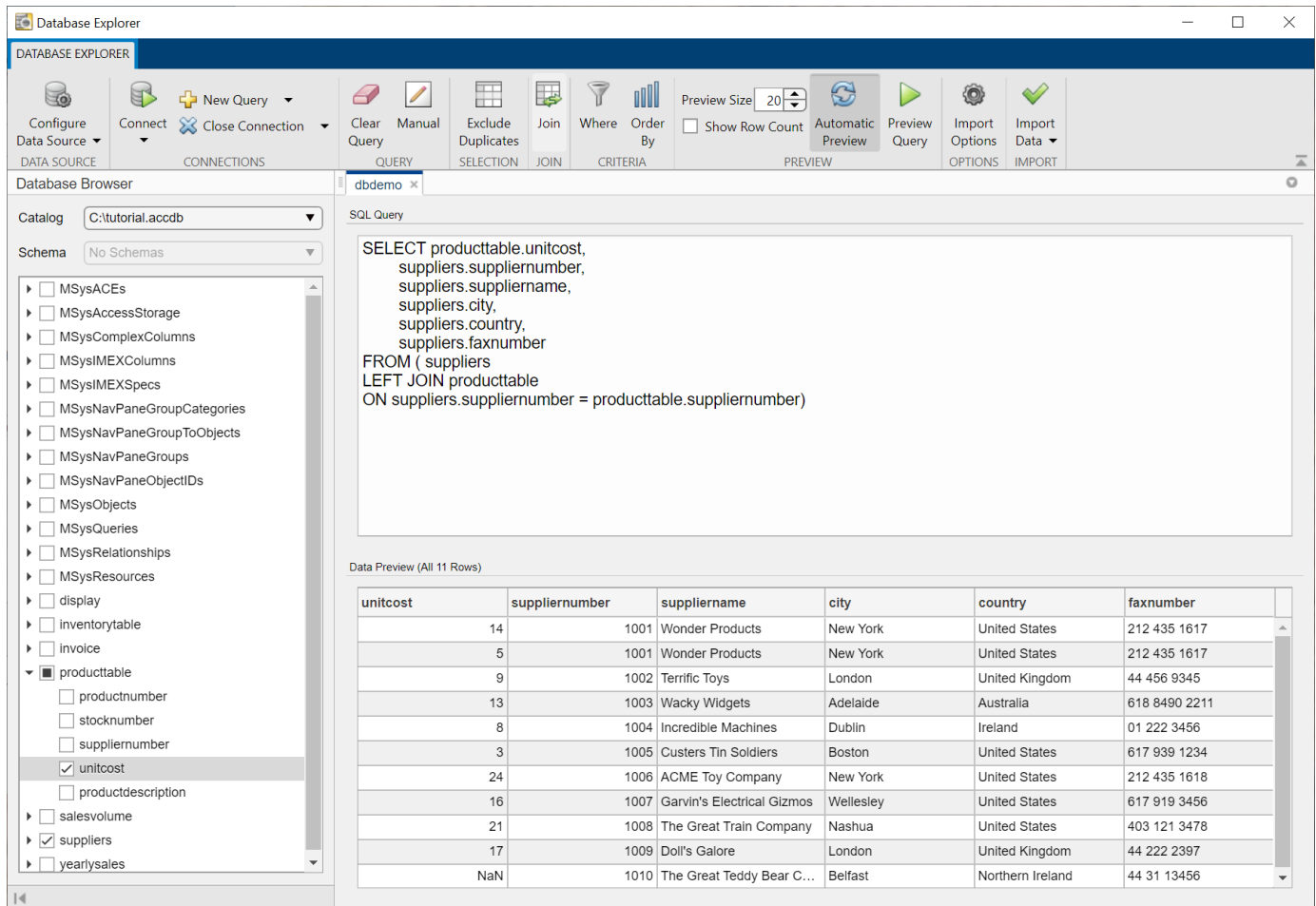
In the left **Column** list, select the name of the shared column **suppliernumber**. In the right **Table** list, select the name **producttable** as the table to join. Select the name of the shared column **suppliernumber** in this table in the right **Column** list.

In the **Add** section, click **Add Join**. The Database Explorer app creates an inner join by default. In the **Edit** section, click **Left** to change the join from an inner join to a left join. The **Join Diagram** pane displays a pictorial representation of the join between the selected tables. The **SQL Query** pane updates the SQL query with the new join. The **Data Preview** pane reflects the results of the updated SQL query.

In the **Close** section, click **Close Join** to close the **Join** tab.

Increase the number of rows displayed in the **Data Preview** pane. In the **Preview** section, enter 20 in the **Preview Size** box.

In the tree view of the **Database Browser** pane, select **unitcost** under **producttable**. The **Data Preview** pane updates with a new column.



The NaN value in the **unitcost** column indicates that the corresponding supplier does not supply products.

17	1009	Doll's Galore	London	United Kingdom	44 222 2397
NaN	1010	The Great Teddy Bear C...	Belfast	Northern Ireland	44 31 13456

Add filter criteria to the SQL query. In the **Criteria** section, click **Where** to display the **Where** tab in the toolstrip.


Filter the SQL query results for products with a unit cost greater than \$10. In the **Add** section, in the **Column** list, select the column name `producttable.unitcost`. Select the > operator for the filter in the **Operator** list. Enter 10 in the **Value** list. Click **Add Filter**.

Note If you enter filters using the LIKE or NOT LIKE operators, then enter the value in single quotes to represent a string.

The **SQL Query** and **Data Preview** panes display the updated query results based on the new filter with the WHERE condition.

Change the value of the filter from 10 to 20. Click **Update Filter**. The **SQL Query** and **Data Preview** panes update with the results of the modified query.

In the **Close** section, click **Close Where** to close the **Where** tab.

Import all SQL query results into the MATLAB Workspace. In the **Import** section, click . In the Import Data dialog box, enter the name `data` for the MATLAB Workspace variable, and click **OK**. The MATLAB Workspace displays the table `data`.

Display the SQL query results at the command line.

`data`

`data =`

2×6 table

suppliernumber	suppliername	city	country	faxnumber	unitcost
1008	'The Great Train Company'	'Nashua'	'United States'	'403 121 3478'	21
1006	'ACME Toy Company'	'New York'	'United States'	'212 435 1618'	24

Find the maximum product price.

`max(data.unitcost)`

`ans =`

24

In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

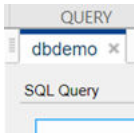
Sort Query Results

Connect to a Microsoft Access database using the Database Explorer app. Create a simple SQL query and sort the results by the data in one column. The Database Explorer app previews query results by default. Then, import the sorted data into the MATLAB workspace.

Set up the data source for the `tutorial.accdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

In the **Connections** section of the **Database Explorer** tab, from the **Connect** list, select the data source for connection. The connection dialog box opens. Leave the user name and password blank, and click **Connect**.

The Database Explorer app creates a connection to the Microsoft Access database. The **Database Browser** pane displays the available tables in the database. The data source tab, which is named **dbdemo**, appears to the right of the **Database Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.



In the **Database Browser** pane, select the **inventorytable** table. The **SQL Query** pane displays the SQL query that selects all columns and rows from this table. The **Data Preview** pane displays the first 10 rows of the query results.

Sort the results of the SQL query. In the **Criteria** section, click **Order By** to display the **Order By** tab in the toolstrip.

In the **Add** section, in the **Column** list, select the **price** column. Click **Add Sort**.

In the **Edit** section, click **Descending** to sort the prices in decreasing order. The **Data Preview** pane displays the updated query results with sorted prices.

The screenshot shows the 'ORDER BY' tab in the SQL Query toolstrip. The 'Column' dropdown is set to 'price' and 'Descending' is selected. The 'SQL Query' pane shows the following query:


```
SELECT *
FROM inventorytable
ORDER BY price DESC
```

The 'Data Preview (First 10 Rows)' pane shows the following data:

productnumber	quantity	price	inventorydate
10		723	24 2012-03-14 13:13:09
4		2580	21 2013-06-08 14:24:33
3		356	17 2014-05-14 07:14:28
7		6034	16 2014-08-06 08:38:00
13		1700	14.5000 2009-05-24 10:58:59
1		1700	14.5000 2014-09-23 09:38:34
9		2339	13 2011-02-09 12:50:59
2		1200	9 2014-07-08 22:50:45
6		4540	8 2013-12-25 19:45:00
8		8350	5 2011-06-18 11:45:35

In the **Close** section, click **Close Order By** to close the **Order By** tab.



Import all SQL query results into the MATLAB Workspace. In the **Import** section, click . In the Import Data dialog box, enter the name `data` for the MATLAB Workspace variable, and click **OK**. The MATLAB Workspace displays the table `data`.

In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

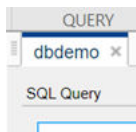
Filter Query Results

Connect to a Microsoft Access database using the Database Explorer app. Create a simple SQL query and filter the results. Use a text filter to retrieve specific rows of data. The Database Explorer app previews query results by default. Then, import the filtered data into the MATLAB workspace.

Set up the data source for the `tutorial.accdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

In the **Connections** section of the **Database Explorer** tab, from the **Connect** list, select the data source for connection. The connection dialog box opens. Leave the user name and password blank, and click **Connect**.

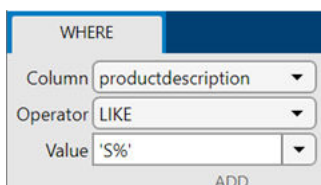
The Database Explorer app creates a connection to the Microsoft Access database. The **Database Browser** pane displays the available tables in the database. The data source tab, which is named **dbdemo**, appears to the right of the **Database Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.



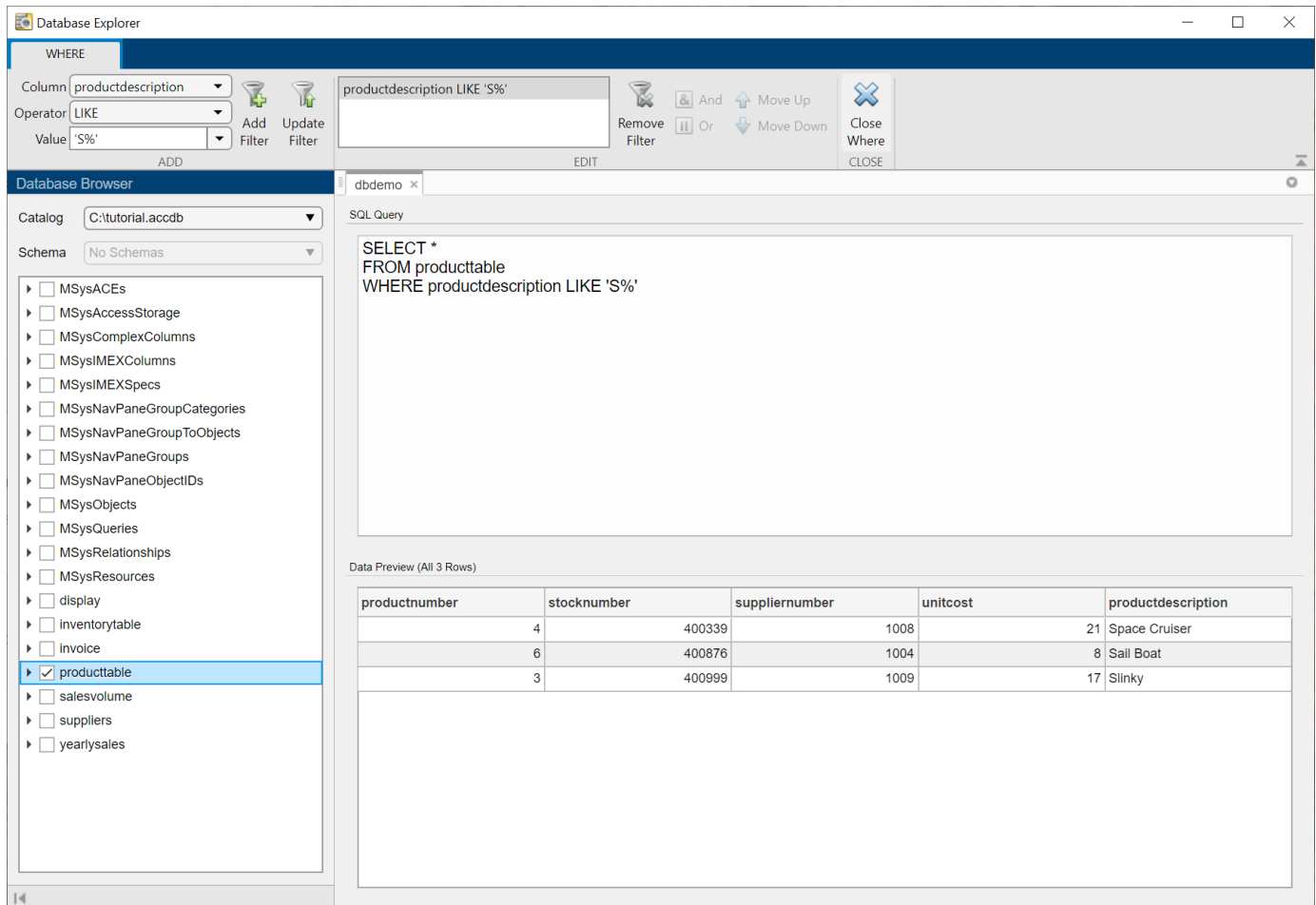
In the **Database Browser** pane, select the **producttable** table. The **SQL Query** pane displays the SQL query that selects all columns and rows from this table. The **Data Preview** pane displays the first 10 rows of the query results.

Add filter criteria to the SQL query. In the **Criteria** section, click **Where** to display the **Where** tab in the toolstrip.

Filter for products with a product description that starts with the letter S. In the **Add** section, in the **Column** list, select `productdescription`. In the **Operator** list, select `LIKE`. To filter for text, enclose the text in single quotes. In the **Value** list, enter `'S%'`.



Click **Add Filter**. The **Data Preview** pane displays three rows of data. The product description in each row starts with the letter S.




The screenshot shows the Database Explorer interface. The WHERE tab is active, showing a filter for the 'productdescription' column with the operator 'LIKE' and the value 'S%'. The SQL Query pane contains the following query:

```
SELECT *
FROM producttable
WHERE productdescription LIKE 'S%'
```

The Data Preview pane displays the following data:

productnumber	stocknumber	suppliernumber	unitcost	productdescription
4	400339	1008	21	Space Cruiser
6	400876	1004	8	Sail Boat
3	400999	1009	17	Slinky

In the **Close** section, click **Close Where** to close the **Where** tab.

Import all SQL query results into the MATLAB Workspace. In the **Import** section, click . In the Import Data dialog box, enter the name `data` for the MATLAB Workspace variable, and click **OK**. The MATLAB Workspace displays the table `data`.

In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

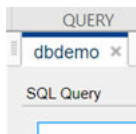
Remove Duplicate Rows from Query Results

Connect to a Microsoft Access database using the Database Explorer app. Create a simple SQL query and remove duplicate rows from the query results. The Database Explorer app previews query results by default. After removing duplicates, import the data into the MATLAB workspace.

Set up the data source for the `tutorial.accdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

In the **Connections** section of the **Database Explorer** tab, from the **Connect** list, select the data source for connection. The connection dialog box opens. Leave the user name and password blank, and click **Connect**.

The Database Explorer app creates a connection to the Microsoft Access database. The **Database Browser** pane displays the available tables in the database. The data source tab, which is named **dbdemo**, appears to the right of the **Database Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.

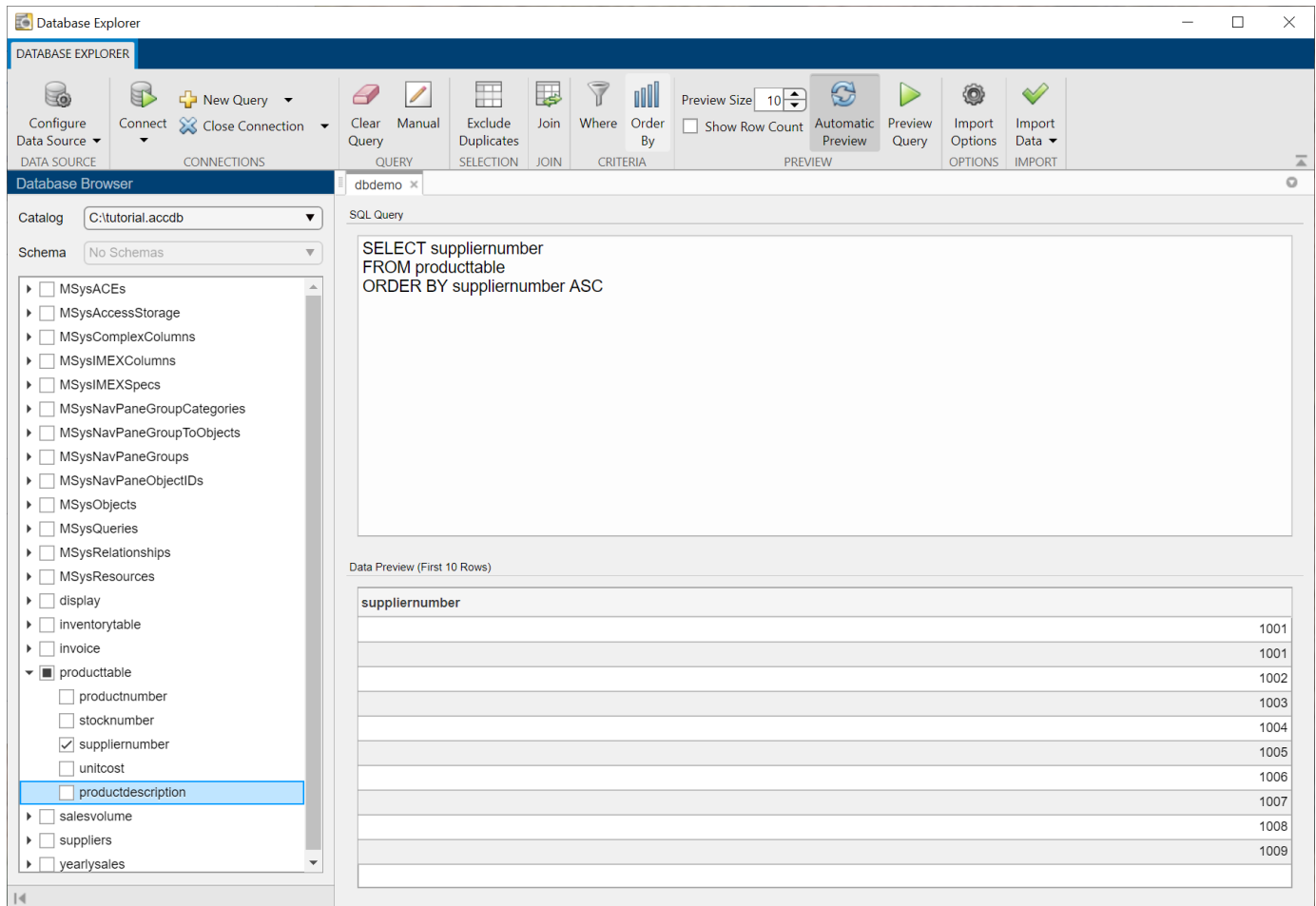


In the **Database Browser** pane, select the **producttable** table. Clear all the boxes for columns in the **producttable** table except for **suppliernumber**. The **SQL Query** pane displays the SQL query that selects the **suppliernumber** column from this table. The **Data Preview** pane displays the first 10 rows of the query results.

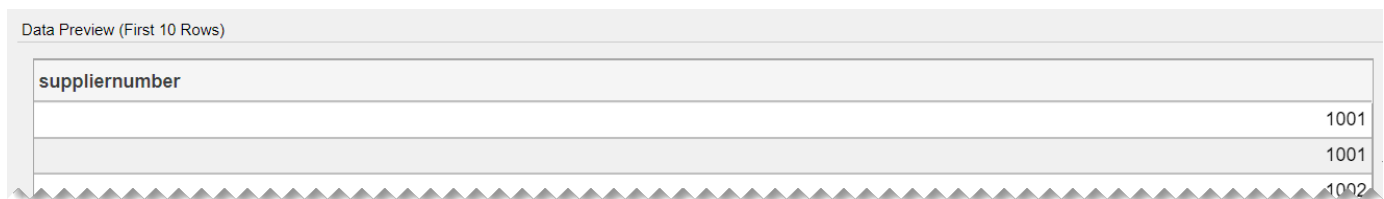
Sort the results of the SQL query. In the **Criteria** section, click **Order By** to display the **Order By** tab in the toolstrip. In the **Add** section, in the **Column** list, select the **suppliernumber** column, and click **Add Sort**.

In the **Close** section, click **Close Order By** to close the **Order By** tab.

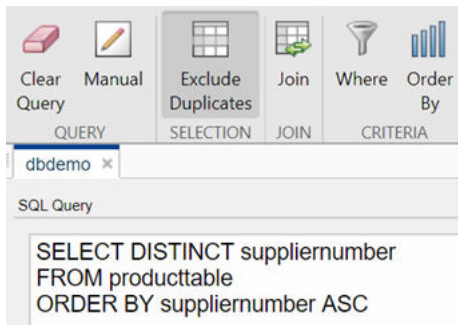
The **Data Preview** pane displays the rows sorted in increasing order, which is the default order.



The **Data Preview** pane shows the duplicate supplier number 1001.



In the **Selection** section, click **Exclude Duplicates** to remove duplicate rows in the **Data Preview** pane. The Database Explorer App adds the SQL statement **DISTINCT** to the query in the **SQL Query** pane. This statement removes duplicate rows from the query results.



The **Data Preview** pane displays unique rows only.


Database Explorer

SQL Query

```
SELECT DISTINCT suppliernumber
FROM producttable
ORDER BY suppliernumber ASC
```

Data Preview (All 9 Rows)

suppliernumber	
	1001
	1002
	1003
	1004
	1005
	1006
	1007
	1008
	1009

Import all SQL query results into the MATLAB Workspace. In the **Import** section, click . In the Import Data dialog box, enter the name **data** for the MATLAB Workspace variable, and click **OK**. The MATLAB Workspace displays the table data.

In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

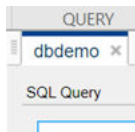
Enter SQL Query Manually

Connect to a Microsoft Access database using the Database Explorer app. Enter an SQL query manually or paste an existing SQL query into the **SQL Query** pane. Then, import the query results into the MATLAB workspace.

Set up the data source for the `tutorial.accdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

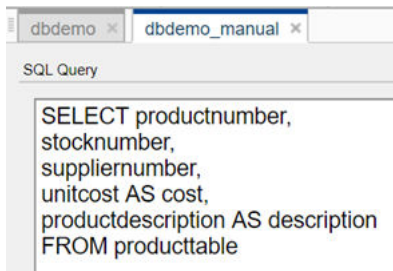
In the **Connections** section of the **Database Explorer** tab, from the **Connect** list, select the data source for connection. The connection dialog box opens. Leave the user name and password blank, and click **Connect**.

The Database Explorer app creates a connection to the Microsoft Access database. The **Database Browser** pane displays the available tables in the database. The data source tab, which is named **dbdemo**, appears to the right of the **Database Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.



In the **Query** section, click **Manual**. A new data source tab appears to the right of the **dbdemo** tab with the name **dbdemo_manual**. The suffix `_manual` attached to the tab name indicates that you are entering an SQL query manually.

Enter an SQL query in the **SQL Query** pane. Here, select all columns and rows from the `producttable` table, and rename the `unitcost` and `productdescription` columns. Use the SQL statement `AS` to create aliases.



In the **Preview** section, click **Preview Query** to preview the query results.


The **Data Preview** pane shows the results of the SQL query. The pane displays the first 10 rows of data by default.

The screenshot shows the Database Explorer window with the following components:

- Toolbar:** Includes buttons for 'Configure Data Source', 'Connect', 'Close Connection', 'Clear Query', 'Manual Query', 'Exclude Duplicates', 'Join', 'Where', 'Order By', 'Preview Size' (set to 10), 'Show Row Count', 'Automatic Preview', 'Preview Query', 'Import Options', and 'Import Data'.
- Database Browser:** Shows the catalog 'C:\tutorial.accdb' and a schema 'No Schemas'. A list of system tables is visible on the left, including 'MSysACEs', 'MSysAccessStorage', 'MSysComplexColumns', 'MSysIMEXColumns', 'MSysIMEXSpecs', 'MSysNavPaneGroupCategories', 'MSysNavPaneGroupToObjects', 'MSysNavPaneGroups', 'MSysNavPaneObjectIDs', 'MSysObjects', 'MSysQueries', 'MSysRelationships', 'MSysResources', 'display', 'inventorytable', 'invoice', 'producttable', 'salesvolume', 'suppliers', and 'yearlysales'.
- SQL Query:** Contains the following query:


```
SELECT productnumber,
stocknumber,
suppliernumber,
unitcost AS cost,
productdescription AS description
FROM producttable
```
- Data Preview (First 10 Rows):** A table showing the results of the query:

productnumber	stocknumber	suppliernumber	cost	description
9	125970	1003	13	Victorian Doll
8	212569	1001	5	Train Set
7	389123	1007	16	Engine Kit
2	400314	1002	9	Painting Set
4	400339	1008	21	Space Cruiser
1	400345	1001	14	Building Blocks
5	400455	1005	3	Tin Soldier
6	400876	1004	8	Sail Boat
3	400999	1009	17	Slinky
10	888652	1006	24	Teddy Bear

Import all SQL query results into the MATLAB Workspace. In the **Import** section, click . In the Import Data dialog box, enter the name `data` for the MATLAB Workspace variable, and click **OK**. The MATLAB Workspace displays the table data.

In the **Connections** section, close the database connection by clicking **Close Connection**.

Note If multiple connections are open, close the database connection of your choice by selecting the corresponding data source from the **Close Connection** list.

Version History

Introduced in R2017b

See Also

Functions

database | exec | fetch | close

Topics

“Connection Options” on page 2-8

“Configure Driver and Data Source” on page 2-14

“Create SQL Queries Using Database Explorer App” on page 4-2

“Join Tables Using Database Explorer App” on page 4-10

“Data Preview Using Database Explorer App” on page 4-14

“Customize Import Options Using Database Explorer App” on page 4-7

“Generate SQL Query and MATLAB Script” on page 4-20

“Modify and Delete Data Sources” on page 4-17

“Database Explorer App Error Messages” on page 3-13

External Websites

SQL Tutorial

exec

Package: database.odbc

(Not recommended) Execute SQL statement and open cursor

Note The `exec` function is not recommended. For SQL statements that return data, use the `fetch` function or the `select` function instead. For other SQL statements, use the `execute` function instead. For details, see “Compatibility Considerations”.

The scrollable cursor functionality has no replacement.

Syntax

```
curs = exec(conn,sqlquery)
curs = exec(conn,sqlquery,Name,Value)
curs = exec(conn,sqlquery,qTimeOut)
```

Description

`curs = exec(conn,sqlquery)` creates the cursor object after executing the SQL statement `sqlquery` for the database connection `conn`.

`curs = exec(conn,sqlquery,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'MaxRows', 10` limits the number of rows to return to 10 before SQL query execution.

`curs = exec(conn,sqlquery,qTimeOut)` uses the timeout value `qTimeOut` for SQL query execution.

Examples

Select Data Using Native ODBC Driver

Use a native ODBC connection to import product data from a Microsoft SQL Server database into MATLAB. Then, determine the highest unit cost among products.

Create an ODBC database connection to a Microsoft SQL Server database with Windows authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, the connection is successful.

```
conn.Message
```

```
ans =  
  
    []
```

Select all data from the table `productTable` using the connection object. Assign the SQL `SELECT` statement to the variable `sqlquery`. The cursor object contains the executed SQL query.

```
sqlquery = 'SELECT * FROM productTable';  
curs = exec(conn,sqlquery)
```

```
curs =
```

```
    cursor with properties:
```

```
        Data: 0  
    RowLimit: 0  
    SQLQuery: 'SELECT * FROM productTable'  
    Message: []  
        Type: 'ODBCCursor Object'  
    Statement: [1x1 database.internal.ODBCStatementHandle]
```

For an ODBC connection, the `Type` property contains `ODBCCursor Object`. For JDBC connections, this property contains `Database Cursor Object`.

Import data from the table into MATLAB.

```
curs = fetch(curs);  
data = curs.Data;
```

Determine the highest unit cost in the table.

```
max(data.unitCost)
```

```
ans =  
  
    24
```

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the database function.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar. The SQL statement can be any valid SQL statement, including nested queries. The SQL statement can be a stored procedure, such as `{call sp_name (parm1,parm2,...)}`. For stored procedures that return one or more result sets, use the `exec` function. For procedures that return output arguments, use `runstoredprocedure`.

For information about the SQL query language, see the SQL Tutorial.

Data Types: `char` | `string`

qTimeout — Timeout value

numeric scalar

Timeout value, specified as a numeric scalar denoting the maximum amount of time, in seconds, that `exec` tries to execute the SQL statement `sqlquery`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `curs = exec(conn,sqlquery,'MaxRows',rowlimit);`

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return before executing the SQL query, specified as the comma-separated pair consisting of `'MaxRows'` and a positive numeric scalar. By default, the `exec` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB from the SQL query execution. For details about this option and other memory management options, see “Data Import Memory Management” on page 5-20.

Data Types: `double`

CursorType — Cursor type

`'forward_only'` (default) | `'scrollable'`

Cursor type, specified as the comma-separated pair consisting of `'CursorType'` and one of the values in this table.

Value	Description
<code>'forward_only'</code>	Create a basic cursor.
<code>'scrollable'</code>	Create a scrollable cursor.

Output Arguments

curs — Database cursor

cursor object

Database cursor, returned as a cursor object.

Limitations

The name-value pair argument `'MaxRows'` has these limitations:

- If you are using Microsoft Access, the native ODBC interface is not supported.
- Not all database drivers support setting the maximum number of rows before query execution. For an unsupported driver, modify your SQL query to limit the maximum number of rows to return. The SQL syntax varies with the driver. For details, consult the driver documentation.

Tips

- The order of records in your database does not remain constant. Sort data using the SQL `ORDER BY` command in your `sqlquery` statement.
- For Microsoft Excel, tables in `sqlquery` are Excel worksheets. By default, some worksheet names include a \$ symbol. To select data from a worksheet with this name format, use an SQL statement of the form `SELECT * FROM "Sheet1$" (or 'Sheet1$')`.
- Before you modify database tables, ensure that the database is not open for editing. If you try to edit the database while it is open, you receive this MATLAB error:

```
[Vendor][ODBC Driver] The database engine could not lock
table 'TableName' because it is already in use by
another person or process.
```

- The PostgreSQL database management system supports multidimensional fields, but SQL `SELECT` statements fail when retrieving these fields unless you specify an index.
- Some databases require that you include a symbol, such as #, before and after a date in a query, as follows:

```
curs = exec(conn, 'SELECT * FROM mydb WHERE mydate > #03/05/2005#')
```

Alternative Functionality

App

The `exec` function executes SQL statements using the command line. To execute SQL statements interactively, use the **Database Explorer** app.

Version History

Introduced before R2006a

R2018b: `exec` function is not recommended

Not recommended starting in R2018b

The `exec` function is not recommended. For SQL statements that return data, use the `fetch` function with the connection object or the `select` function instead. For other SQL statements, use the `execute` function instead. Some differences between the workflows might require updates to your code.

There are no plans to remove the `exec` function at this time.

Update Code

Use the `fetch` function with the connection object to import data from a database in one step.

In prior releases, you wrote multiple lines of code to create the cursor object and import data. For example:

```
curs = exec(conn,sqlquery);  
curs = fetch(curs);  
results = curs.Data;  
close(curs)
```

Now you can import data in one step using the `fetch` function.

```
results = fetch(conn,sqlquery);
```

You can also import data in one step using the `select` function.

```
data = select(conn,selectquery);
```

The scrollable cursor functionality has no replacement.

See Also

`close` | `database` | `fetch` | `setdbprefs` | `select`

Topics

“Import Data from Database Table Using `sqlread` Function” on page 5-58

“Call Stored Procedure That Returns Data” on page 5-17

“Roll Back Data in Database” on page 5-2

“Change Database Connection Catalog” on page 5-4

“Create Table and Add Column” on page 5-5

“Run Custom Database Function” on page 5-19

“Data Import Memory Management” on page 5-20

“Data Retrieval Restrictions” on page 1-5

External Websites

SQL Tutorial

exec

Execute SQL statement using SQLite connection

Note The `exec` function will be removed in a future release. Use the `execute` function to manage the SQLite database instead.

Syntax

```
exec(conn,sqlquery)
```

Description

`exec(conn,sqlquery)` performs database operations on an SQLite database file by executing the SQL statement `sqlquery` for the SQLite connection `conn` using the MATLAB interface to SQLite. For example, use this syntax to create database tables in the SQLite database file. To import data into MATLAB from the SQLite database file, use the `fetch` function.

Examples

Create Table Using MATLAB Interface to SQLite

Using the MATLAB® Interface to SQLite, create a table in a new SQLite database file.

Create the SQLite connection `conn` to the new SQLite database file `tutorial.db`. Specify the file name in the current folder.

```
dbfile = fullfile(pwd,'tutorial.db');  
conn = sqlite(dbfile,'create');
```

Create the table `inventoryTable` using `exec`.

```
createInventoryTable = ['create table inventoryTable ' ...  
    '(productNumber NUMERIC, Quantity NUMERIC, ' ...  
    'Price NUMERIC, inventoryDate VARCHAR)'];
```

```
exec(conn,createInventoryTable)
```

`inventoryTable` is an empty table in `tutorial.db`.

To insert data into the database file, use the `insert` function.

Close the SQLite connection.

```
close(conn)
```


Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar. The SQL statement can be any valid SQL statement, including nested queries. The SQL statement can be a stored procedure, such as `{call sp_name (parm1,parm2,...)}`. For stored procedures that return one or more result sets, use the `exec` function. For procedures that return output arguments, use `runstoredprocedure`.

For information about the SQL query language, see the SQL Tutorial.

Data Types: `char` | `string`

Version History

Introduced in R2016a

See Also

`close` | `sqlite` | `insert` | `fetch`

Topics

“Import Data Using MATLAB Interface to SQLite” on page 5-32

“Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5

External Websites

SQL Tutorial

execute

Package: database.odbc

Execute SQL statement using relational database connection

Syntax

```
execute(conn,sqlquery)
execute(conn,pstmt)
```

Description

`execute(conn,sqlquery)` executes an SQL query that contains a non-SELECT SQL statement by using the relational database connection.

`execute(conn,pstmt)` executes an SQL prepared statement that contains a non-SELECT SQL statement by using the relational database connection.

Examples

Execute Non-SELECT SQL Statement

Using a relational database connection, create and execute a non-SELECT SQL statement that deletes a database table.

This example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';
sqlwrite(conn,tablename,patients)
```

Import the data from the `patients` database table.

```
data = sqlread(conn,tablename);
```

Delete the `patients` database table using the `execute` function.

```
sqlquery = ['DROP TABLE ' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Call Stored Procedure Without Input and Output Arguments

Working with a Microsoft SQL Server database, run a stored procedure by using the native ODBC database connection `conn`.

Define a stored procedure named `create_table` that creates a table named `test_table` by executing the following code. This procedure has no input or output arguments. The code assumes that you are using a Microsoft SQL Server database.

```
CREATE PROCEDURE create_table
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    CREATE TABLE test_table
    (
        CATEGORY_ID    INTEGER    IDENTITY PRIMARY KEY,
        CATEGORY_DESC  CHAR(50)    NOT NULL
    );

END
GO
```

Connect to the Microsoft SQL Server database. This code assumes that you are connecting to a data source named `MS SQL Server` with a user name and password.

```
conn = database('MS SQL Server', 'username', 'pwd');
```

Call the stored procedure `create_table`.

```
execute(conn, 'create_table')
```

Insert Data Using SQL Prepared Statement

Create an SQL prepared statement to insert data from MATLAB® into a Microsoft® SQL Server® database using a JDBC database connection. Use the `INSERT SQL` statement for the SQL query. Execute the SQL prepared statement and display the results.

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MSSQLServerJDBCAuth';
conn = database(datasource, '', '');
```

Import data from the database using the `sql read` function. Display the last few rows of data in the database table `inventoryTable`.

```
tablename = "inventoryTable";
data = sqlread(conn,tablename);
tail(data,3)
```

```
ans=3x4 table
   productNumber   Quantity   Price   inventoryDate
   _____   _____   _____   _____
           11           567           0   {'2012-09-11 00:30:24'}
           12          1278           0   {'2010-10-29 18:17:47'}
           13          1700          14.5   {'2009-05-24 10:58:59'}
```

Create an SQL prepared statement for inserting data using the JDBC database connection. The question marks in the INSERT SQL statement indicate it is an SQL prepared statement. This statement inserts data from MATLAB into the database table `inventoryTable`.

```
query = "INSERT INTO inventoryTable VALUES(?,?,?,?)";
pstmt = databasePreparedStatement(conn,query)
```

```
pstmt =
  SQLPreparedStatement with properties:
    SQLQuery: "INSERT INTO inventoryTable values(?,?,?,?)"
  ParameterCount: 4
  ParameterTypes: ["numeric" "numeric" "numeric" "string"]
  ParameterValues: {[] [] [] []}
```

`pstmt` is an `SQLPreparedStatement` object with these properties:

- `SQLQuery` — SQL prepared statement query
- `ParameterCount` — Parameter count
- `ParameterTypes` — Parameter types
- `ParameterValues` — Parameter values

Bind parameter values in the SQL prepared statement. Select all parameters in the SQL prepared statement using their numeric indices. Specify the values to bind for the product number, quantity, price, and inventory date. Match the format of dates in the database. The `bindParamValues` function updates the values in the `ParameterValues` property of the `pstmt` object.

```
selection = [1 2 3 4];
values = {20,1000,55,"2019-04-25 00:00:00.000"};
pstmt = bindParamValues(pstmt,selection,values)
```

```
pstmt =
  SQLPreparedStatement with properties:
    SQLQuery: "INSERT INTO inventoryTable values(?,?,?,?)"
  ParameterCount: 4
  ParameterTypes: ["numeric" "numeric" "numeric" "string"]
  ParameterValues: {[20] [1000] [55] ["2019-04-25 00:00:00.000"]}
```

Insert data from MATLAB into the database using the bound parameter values. Execute the SQL INSERT statement using the `execute` function.

```
execute(conn,pstmt)
```

Display the inserted data in the database table `inventoryTable`. The last row in the table contains the inserted data.

```
tablename = "inventoryTable";
data = sqlread(conn,tablename);
tail(data,4)
```

```
ans=4x4 table
  productNumber  Quantity  Price  inventoryDate
  _____  _____  _____  _____
      11           567         0  {'2012-09-11 00:30:24' }
      12          1278         0  {'2010-10-29 18:17:47' }
      13          1700        14.5  {'2009-05-24 10:58:59' }
      20          1000         55  {'2019-04-25 00:00:00.000' }
```

Close the SQL prepared statement and database connection.

```
close(pstmt)
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the database function.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar. The SQL statement can be any valid non-SELECT SQL statement.

The SQL statement can be a stored procedure that does not return any result sets. For stored procedures that return one or more result sets, use the `fetch` function. For procedures that return output arguments, use the `runstoredprocedure` function.

For information about the SQL query language, see the SQL Tutorial.

Example: 'DROP TABLE patients'

Data Types: char | string

pstmt — SQL prepared statement

SQLPreparedStatement object

SQL prepared statement, specified as an SQLPreparedStatement object.

Version History

Introduced in R2018b

See Also

close | database | fetch | sqlread | runstoredprocedure | databasePreparedStatement | bindParamValues | close

Topics

“Create Table and Add Column” on page 5-5

“Delete Data from Databases” on page 5-6

“Roll Back Data in Database” on page 5-2

“Import Data Using SQL Prepared Statement with Multiple Parameter Values” on page 5-75

“SQL Prepared Statement Error Messages” on page 3-15

External Websites

SQL Tutorial

execute

Execute SQL statement using SQLite database connection

Syntax

```
execute(conn,sqlquery)
```

Description

`execute(conn,sqlquery)` executes an SQL query that contains a non-SELECT SQL statement by using the SQLite database connection with the MATLAB interface to SQLite.

Examples

Execute Non-SELECT SQL Statement

Using an SQLite database connection and the MATLAB® interface to SQLite, create and execute a non-SELECT SQL statement that creates a temporary view in the database and imports its contents.

Create an SQLite database connection to the SQLite database file `tutorial.db`.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Create an SQL statement that creates a temporary view named `PRODUCTNAMES`. The view selects all product names using the `productDescription` column of the `productTable` database table. Execute the CREATE SQL statement.

```
sqlquery = strcat("CREATE TEMP VIEW PRODUCTNAMES AS", ...
    " SELECT productDescription FROM productTable");
execute(conn,sqlquery)
```

Import the product names using the new temporary view and display the first three names.

```
sqlquery = "SELECT * FROM PRODUCTNAMES";
results = fetch(conn,sqlquery);
head(results,3)
```

```
productDescription
-----
"Victorian Doll"
"Train Set"
"Engine Kit"
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar. The SQL statement can be any valid non-SELECT SQL statement. For information about the SQL query language, see the SQL Tutorial.

Example: "DROP VIEW PRODUCTNAMES"

Data Types: `char` | `string`

Version History

Introduced in R2022a

See Also

Objects

`sqlite`

Functions

`fetch` | `sqlwrite` | `close`

Topics

"Insert Data into SQLite Database Table" on page 5-36

"Create Table and Add Column in SQLite Database" on page 5-38

"Delete Data from SQLite Database" on page 5-39

"Roll Back Data in SQLite Database" on page 5-41

External Websites

SQL Tutorial

SQLite Home Page

fastinsert

Package: database.odbcc

(To be removed) Add MATLAB data to database tables

Note The `fastinsert` function will be removed in a future release. Use the `sqlwrite` function instead. For details, see “Compatibility Considerations”.

Syntax

```
fastinsert(conn,tablename,colnames,data)
```

Description

`fastinsert(conn,tablename,colnames,data)` exports data from the MATLAB workspace and inserts it into an existing database table using the database connection `conn`. You can specify the database table name and column names, and specify the data for insertion into the database.

You do not specify the type of data you are exporting. The data is exported in its current MATLAB format.

Examples

Insert Row into Table Using ODBC Driver

First, connect to the Microsoft® SQL Server® database. Then, export data from MATLAB® into the database and close the database connection.

Create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Select and display all rows in the table sorted by the product number using the `select` function.

```
selectquery = 'SELECT * FROM productTable ORDER BY productNumber';
```

```
data = select(conn,selectquery)
```

```
data =
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
1	4.0035e+05	1001	14	'Building Blocks'
2	4.0031e+05	1002	9	'Painting Set'
3	4.01e+05	1009	17	'Slinky'

Store the column names of `productTable` in a cell array.

```
tablename = 'productTable';
colnames = {'productNumber','stockNumber','supplierNumber', ...
            'unitCost','productDescription'};
```

Store the data for the insert in a cell array that contains these values:

- `productNumber` equal to 4
- `stockNumber` equal to 500565
- `supplierNumber` equal to 1010
- `unitCost` equal to \$20
- `productDescription` equal to 'Cooking Set'

Then, convert the cell array to a table.

```
insertdata = {4,500565,1010,20,'Cooking Set'};
insertdata = cell2table(insertdata,'VariableNames',colnames)
```

```
insertdata =
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
4	5.0057e+05	1010	20	'Cooking Set'

Insert data into the table.

```
fastinsert(conn,tablename,colnames,insertdata)
```

Select and display all rows in the table again.

```
data = select(conn,selectquery)
```

```
data =
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
1	4.0035e+05	1001	14	'Building Blocks'
2	4.0031e+05	1002	9	'Painting Set'
3	4.01e+05	1009	17	'Slinky'

```
4          5.0057e+05    1010          20          'Cooking Set'
```

A new row appears in the `productTable` with data from `insertdata`.

Close the database connection.

```
close(conn)
```

Insert Multiple Rows into Table

First, connect to the Microsoft® SQL Server® database. Then, export multiple rows of data from MATLAB® into the database and close the database connection.

Create an ODBC database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Select and display data in the table `inventoryTable`. Import data using the `select` function.

```
selectquery = 'SELECT * FROM inventoryTable';
data = select(conn,selectquery)
```

```
data =
```

<u>productNumber</u>	<u>Quantity</u>	<u>Price</u>	<u>inventoryDate</u>
1	1700	15	'2014-09-23'
2	1200	9	'2014-07-08'
3	356	17	'2014-05-14'
4	2580	21	'2013-06-08'
5	9000	3	'2012-09-14'
6	4540	8	'2013-12-25'
7	6034	16	'2014-08-06'
8	8350	5	'2011-06-18'
9	2339	13	'2011-02-09'
10	723	24	'2012-03-14'

Assign multiple rows of data to the cell array `insertdata`. Each row contains data for the columns in `inventoryTable`. The first row of data contains:

- Product number is 11
- Quantity is 125
- Price is \$23.00
- Inventory date is the current date

```
insertdata = {11,125,23.00,datestr(now,'yyyy-mm-dd'); ...
             12,1160,14.7,datestr(now,'yyyy-mm-dd'); ...
             13,150,54.5,datestr(now,'yyyy-mm-dd')};
```

Store the column names of `inventoryTable` in a cell array.

```
tablename = 'inventoryTable';
colnames = {'productNumber','Quantity','Price','inventoryDate'};
```

Insert data into the table.

```
fastinsert(conn,tablename,colnames,insertdata)
```

Select and display data in the table `inventoryTable` again.

```
data = select(conn,selectquery)
```

data =

productNumber	Quantity	Price	inventoryDate
1	1700	15	'2014-09-23'
2	1200	9	'2014-07-08'
3	356	17	'2014-05-14'
4	2580	21	'2013-06-08'
5	9000	3	'2012-09-14'
6	4540	8	'2013-12-25'
7	6034	16	'2014-08-06'
8	8350	5	'2011-06-18'
9	2339	13	'2011-02-09'
10	723	24	'2012-03-14'
11	125	23	'2016-11-02'
12	1160	15	'2016-11-02'
13	150	55	'2016-11-02'

Three new rows appear in `inventoryTable` with data from `insertdata`.

Close the database connection.

```
close(conn)
```

Insert Numeric Data into Table

First, connect to the Microsoft® SQL Server® database. Then, export numeric data from MATLAB® into the database and close the database connection.

Create an ODBC database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the Message property is empty, the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Define the numeric matrix numdata that contains sales volume data.

```
numdata = [777666,0,350,400,450,250,450,500,515,235,100,300,600];
```

Select and display data in the salesVolume table before insertion. Import data using the select function.

```
selectquery = 'SELECT * FROM salesVolume';
data = select(conn,selectquery)
```

```
data =
```

StockNumber	January	February	March	April	May	June	July	August	Sep
1.2597e+05	1400	1100	981	882	794	752	654	773	800
2.1257e+05	2400	1721	1414	1191	983	825	731	653	720
3.8912e+05	1800	1200	890	670	550	450	400	410	400
4.0031e+05	3000	2400	1800	1500	1200	900	700	650	1600
4.0034e+05	4300	0	2600	1800	1600	1550	895	700	750
4.0035e+05	5000	3500	2800	2300	1700	1400	1000	900	1600
4.0046e+05	1200	900	800	500	399	345	300	175	700
4.0088e+05	3000	2400	1500	1500	1300	1100	900	867	920
4.01e+05	3000	1500	1000	900	750	700	400	350	500
8.8865e+05	0	900	821	701	689	621	545	421	490
4.0814e+05	6000	3100	8800	2300	1700	1400	1000	900	1600
2.1046e+05	1800	9700	800	500	3997	349	300	175	700
4.7082e+05	3100	9400	1540	1500	1350	1190	900	867	920
5.101e+05	235	1800	1040	900	750	700	400	350	500
8.9975e+05	123	1700	823	701	689	621	545	421	490

Store the column names of salesVolume in a cell array.

```
tablename = 'salesVolume';
colnames = {'stockNumber','January','February','March','April','May', ...
            'June','July','August','September','October','November', ...
            'December'};
```

Insert data into the table.

```
fastinsert(conn,tablename,colnames,numdata)
```

Select and display data in the salesVolume table again.

```
data = select(conn,selectquery)
```

```
data =
```

StockNumber	January	February	March	April	May	June	July	August	Sep
1.2597e+05	1400	1100	981	882	794	752	654	773	800
2.1257e+05	2400	1721	1414	1191	983	825	731	653	720
3.8912e+05	1800	1200	890	670	550	450	400	410	400
4.0031e+05	3000	2400	1800	1500	1200	900	700	650	1600
4.0034e+05	4300	0	2600	1800	1600	1550	895	700	750
4.0035e+05	5000	3500	2800	2300	1700	1400	1000	900	1600
4.0046e+05	1200	900	800	500	399	345	300	175	700
4.0088e+05	3000	2400	1500	1500	1300	1100	900	867	920
4.01e+05	3000	1500	1000	900	750	700	400	350	500
8.8865e+05	0	900	821	701	689	621	545	421	490
4.0814e+05	6000	3100	8800	2300	1700	1400	1000	900	1600
2.1046e+05	1800	9700	800	500	3997	349	300	175	700
4.7082e+05	3100	9400	1540	1500	1350	1190	900	867	920
5.101e+05	235	1800	1040	900	750	700	400	350	500
8.9975e+05	123	1700	823	701	689	621	545	421	490
7.7767e+05	0	350	400	450	250	450	500	515	220

A new row appears in `salesVolume` with data from `numdata`.

Close the database connection.

```
close(conn)
```

Insert and Commit Data in Table

First, connect to the Microsoft® SQL Server® database. Then, export data from MATLAB® into the database and commit the insert transaction. Close the database connection.

Create an ODBC database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. Use the name-value pair argument `AutoCommit` to specify manually committing transactions to the database.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', 'AutoCommit', 'off');
```

Check the database connection. If the `Message` property is empty, the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Insert the cell array `data` into the table `inventoryTable` with column names `colnames`.

```
data = {157,358,740.00,datestr(now,'yyyy-mm-dd HH:MM:SS')};
colnames = {'productNumber','Quantity','Price','inventoryDate'};
tablename = 'inventoryTable';
```

```
fastinsert(conn,tablename,colnames,data)
```

Commit the insert transaction.

```
commit(conn)
```

Close the database connection.

```
close(conn)
```

Insert Boolean Data into Table

First, connect to the Microsoft® SQL Server® database. Then, export Boolean data from MATLAB® into the database. Close the database connection.

Create an ODBC database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource,'','');
```

This database contains the table Invoice with these columns:

- InvoiceNumber
- InvoiceDate
- productNumber
- Paid
- Receipt

Check the database connection. If the Message property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Display the data in the Invoice table before insertion.

```
selectquery = 'SELECT * FROM Invoice';
data = select(conn,selectquery)
```

```
data =
```

```
10×5 table
```

InvoiceNumber	InvoiceDate	ProductNumber	Paid	Receipt
---------------	-------------	---------------	------	---------

2101	'2010-08-01 00:00:00.000'	1	false	[8000×1 uint8]
3546	'2010-03-01 00:00:00.000'	2	true	[8000×1 uint8]
33116	'2011-05-15 00:00:00.000'	3	true	[8000×1 uint8]
34155	'2011-07-12 00:00:00.000'	4	false	[8000×1 uint8]
34267	'2011-07-22 00:00:00.000'	5	true	[8000×1 uint8]
37197	'2011-09-03 00:00:00.000'	6	true	[8000×1 uint8]
37281	'2011-09-21 00:00:00.000'	7	false	[8000×1 uint8]
41011	'2011-12-12 00:00:00.000'	8	true	[8000×1 uint8]
61178	'2012-01-15 00:00:00.000'	9	false	[8000×1 uint8]
62145	'2012-01-23 00:00:00.000'	10	true	[8000×1 uint8]

Create the variable `insertdata` as a structure containing the invoice number 2105, product number 11, and the Boolean data `false` to signify unpaid. Boolean data is represented as the MATLAB® data type `logical`. This code assumes that the receipt image is missing.

```
insertdata.InvoiceNumber{1} = 2105;
insertdata.InvoiceDate{1} = datestr(now, 'yyyy-mm-dd HH:MM:SS');
insertdata.productNumber{1} = 11;
insertdata.Paid{1} = false;
```

Insert the paid invoice data into the `Invoice` table with column names `colnames` using the database connection.

```
colnames = {'InvoiceNumber'; 'InvoiceDate'; 'productNumber'; 'Paid'};
tablename = 'Invoice';
```

```
fastinsert(conn, tablename, colnames, insertdata)
```

View the new record in the database to verify that the `Paid` column value is Boolean. In some databases, the MATLAB® logical value `false` shows as a Boolean `false`, `No`, or a cleared check box.

```
data = select(conn, selectquery)
```

```
data =
```

```
11×5 table
```

InvoiceNumber	InvoiceDate	ProductNumber	Paid	Receipt
2101	'2010-08-01 00:00:00.000'	1	false	[8000×1 uint8]
3546	'2010-03-01 00:00:00.000'	2	true	[8000×1 uint8]
33116	'2011-05-15 00:00:00.000'	3	true	[8000×1 uint8]
34155	'2011-07-12 00:00:00.000'	4	false	[8000×1 uint8]
34267	'2011-07-22 00:00:00.000'	5	true	[8000×1 uint8]
37197	'2011-09-03 00:00:00.000'	6	true	[8000×1 uint8]
37281	'2011-09-21 00:00:00.000'	7	false	[8000×1 uint8]
41011	'2011-12-12 00:00:00.000'	8	true	[8000×1 uint8]
61178	'2012-01-15 00:00:00.000'	9	false	[8000×1 uint8]
62145	'2012-01-23 00:00:00.000'	10	true	[8000×1 uint8]
2105	'2017-01-04 10:19:42.000'	11	false	' '

The last row contains the Boolean data `false`.

Close the database connection.


```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

colnames — Database table column names

cell array of character vectors | string array

Database table column names, specified as a cell array of one or more character vectors or a string array to denote the columns in the existing database table `tablename`.

Example: {'col1', 'col2', 'col3'}

Data Types: cell | string

data — Data to insert

numeric matrix | cell array | table | dataset | structure

Data to insert, specified as a numeric matrix, cell array, table, dataset array, or structure that contains all data for insertion into the existing database table `tablename`. If `data` is a structure, then field names in the structure must match `colnames`. If `data` is a table or a dataset array, then the variable names in the table or dataset array must match `colnames`.

To insert data into a structure, table, or dataset array, use this special formatting. Each field or variable in a structure, table, or dataset array must be a cell array or double vector. The double vector must be of size `n-by-1`, where `n` is the number of rows to be inserted.

To reduce conversion time, convert dates to serial date numbers using `datenum` before calling `fastinsert`.

Tips

- The value of the `AutoCommit` property in the `connection` object determines whether `fastinsert` automatically commits the data to the database.
 - To view the `AutoCommit` value, access it using the `connection` object; for example, `conn.AutoCommit`.
 - To set the `AutoCommit` value, use the corresponding name-value pair argument in the `database` function.

- To commit the data to the database, use the `commit` function or issue an SQL COMMIT statement using the `exec` function.
- To roll back the data, use `rollback` or issue an SQL ROLLBACK statement using the `exec` function.
- If an error message like the following appears when you run `fastinsert`, the table might be open in edit mode.

```
[Vendor][ODBC Product Driver] The database engine could
not lock table 'TableName' because it is already in use
by another person or process.
```

In this case, close the table in the database and rerun the `fastinsert` function.

Alternative Functionality

To export MATLAB data into a database, you can use the `datainsert` and `insert` functions. For maximum performance, use `datainsert`.

Version History

Introduced before R2006a

R2018a: `fastinsert` function will be removed

Not recommended starting in R2018a

The `fastinsert` function will be removed in a future release. Use the `sqlwrite` function instead. Some differences between the workflows require updates to your code.

Update Code

In prior releases, you exported data from the MATLAB workspace into a database by using the `fastinsert` function and four input arguments. For example:

```
tablename = 'productTable';
colnames = {'productNumber', 'stockNumber', 'supplierNumber', ...
            'unitCost', 'productDescription'};
insertdata = {4,500565,1010,20, 'Cooking Set'};
insertdata = cell2table(insertdata, 'VariableNames', colnames)
fastinsert(conn, tablename, colnames, insertdata)
```

Now the `sqlwrite` function requires only three input arguments.

```
tablename = 'productTable';
colnames = {'productNumber', 'stockNumber', 'supplierNumber', ...
            'unitCost', 'productDescription'};
insertdata = {4,500565,1010,20, 'Cooking Set'};
insertdata = cell2table(insertdata, 'VariableNames', colnames)
sqlwrite(conn, tablename, insertdata)
```

See Also

`sqlwrite` | `commit` | `database` | `insert` | `logical` | `rollback` | `close` | `select`

Topics

“Export Data Using Bulk Insert” on page 5-13

“Replace Existing Data in Database” on page 5-12
“Roll Back Data After Updating Record” on page 5-9
“Data Type Support” on page 1-3

External Websites

SQL Tutorial

fetch

Package: database.odbcc

Import data into MATLAB workspace from execution of SQL statement

Syntax

```
results = fetch(conn,sqlquery)
results = fetch(conn,sqlquery,opts)
results = fetch( ___,Name,Value)
[results,metadata] = fetch( ___ )

results = fetch(conn,pstmt)
results = fetch(conn,pstmt,Name,Value)
```

Description

`results = fetch(conn,sqlquery)` returns all rows of data after executing the SQL statement `sqlquery` for the connection object. `fetch` imports data in batches.

`results = fetch(conn,sqlquery,opts)` customizes options for importing data from an executed SQL query by using the `SQLImportOptions` object.

`results = fetch(___,Name,Value)` specifies additional options using one or more name-value arguments with any of the previous input argument combinations. For example, specify `MaxRows = 5` to import five rows of data.

`[results,metadata] = fetch(___)` also returns the metadata table, which contains metadata information about the imported data.

`results = fetch(conn,pstmt)` returns all rows of data after executing the SQL `SELECT` prepared statement `pstmt` for the connection object.

`results = fetch(conn,pstmt,Name,Value)` specifies additional options using one or more name-value arguments. For example, specify `DataReturnFormat = "structure"` to import data as a structure.

Examples

Import All Data Using connection Object

Import all product data from a Microsoft® SQL Server® database table into MATLAB® by using the connection object. Determine the highest unit cost among products in the table. Then, use a row filter to import only the data for products with a unit cost less than 15.

Create an ODBC database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank username and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the Message property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Import all the data from productTable by using the connection object and SQL query, and display the imported data.

```
sqlquery = 'SELECT * FROM productTable';
results = fetch(conn,sqlquery)
```

```
results =
```

```
15x5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	1.2597e+05	1003	13	{'Victorian Doll' }
8	2.1257e+05	1001	5	{'Train Set' }
7	3.8912e+05	1007	16	{'Engine Kit' }
2	4.0031e+05	1002	9	{'Painting Set' }
4	4.0034e+05	1008	21	{'Space Cruiser' }
1	4.0034e+05	1001	14	{'Building Blocks' }
5	4.0046e+05	1005	3	{'Tin Soldier' }
6	4.0088e+05	1004	8	{'Sail Boat' }
3	4.01e+05	1009	17	{'Slinky' }
10	8.8865e+05	1006	24	{'Teddy Bear' }
11	4.0814e+05	1004	11	{'Convertible' }
12	2.1046e+05	1010	22	{'Hugsy' }
13	4.7082e+05	1012	17	{'Pancakes' }
14	5.101e+05	1011	19	{'Shawl' }
15	8.9975e+05	1011	20	{'Snacks' }

Determine the highest unit cost for all products in the table.

```
max(results.unitCost)
```

```
ans =
```

```
24
```

Now, import the data using a row filter. The filter condition is that unitCost must be less than 15.

```
rf = rowfilter("unitcost");
rf = rf.unitcost < 15;
results = fetch(conn,sqlquery,"RowFilter",rf)
```

```
results =
```

```
7×5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	1.2597e+05	1003	13	{'Victorian Doll' }
8	2.1257e+05	1001	5	{'Train Set' }
2	4.0031e+05	1002	9	{'Painting Set' }
1	4.0034e+05	1001	14	{'Building Blocks' }
5	4.0046e+05	1005	3	{'Tin Soldier' }
6	4.0088e+05	1004	8	{'Sail Boat' }
11	4.0814e+05	1004	11	{'Convertible' }

Close the database connection.

```
close(conn)
```

Import Data from SQL Query Using Import Options

Customize import options when importing data from the results of an SQL query on a database. Control the import options by creating an `SQLImportOptions` object. Then, customize import options for different columns in the SQL query. Import data using the `fetch` function.

This example uses the `employees_database.mat` file, which contains the columns `first_name`, `hire_date`, and `DEPARTMENT_NAME`. The example also uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank username and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load employee information into the MATLAB® workspace.

```
employeeedata = load('employees_database.mat');
```

Create the `employees` and `departments` database tables using the employee information.

```
emps = employeeedata.employees;
depts = employeeedata.departments;
```

```
sqlwrite(conn, 'employees', emps)
sqlwrite(conn, 'departments', depts)
```

Create an `SQLImportOptions` object using an SQL query and the `databaseImportOptions` function. This query retrieves all information for employees who are sales managers or programmers.

```
sqlquery = strcat("SELECT * from employees e join departments d ", ...
    "on (e.department_id = d.department_id) WHERE ", ...
    "(job_id = 'IT_PROG' or job_id = 'SA_MAN')");
opts = databaseImportOptions(conn, sqlquery)
```

```

opts =
  SQLImportOptions with properties:

    ExcludeDuplicates: false
    VariableNamingRule: 'modify'

    VariableNames: {'employee_id', 'first_name', 'last_name' ... and 13 more}
    VariableTypes: {'double', 'char', 'char' ... and 13 more}
    SelectedVariableNames: {'employee_id', 'first_name', 'last_name' ... and 13 more}
    FillValues: { NaN, '', '' ... and 13 more }
    RowFilter: <unconstrained>

    VariableOptions: Show all 16 VariableOptions

```

Display the current import options for the variables selected in the `SelectedVariableNames` property of the `SQLImportOptions` object.

```

vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)

```

```

varOpts =
  1x16 SQLVariableImportOptions array with properties:

```

```

Variable Options:

```

	(1)	(2)	(3)	(4)	(5)	(6)
Name:	'employee_id'	'first_name'	'last_name'	'email'	'phone_number'	'hire_date'
Type:	'double'	'char'	'char'	'char'	'char'	'char'
MissingRule:	'fill'	'fill'	'fill'	'fill'	'fill'	'fill'
FillValue:	NaN	''	''	''	''	''

To access sub-properties of each variable, use `getoptions`

Change the data types for the `hire_date`, `DEPARTMENT_NAME`, and `first_name` variables using the `setoptions` function. Then, display the updated import options. Because `hire_date` stores date and time data, change the data type of this variable to `datetime`. Because `DEPARTMENT_NAME` designates a finite set of repeating values, change the data type of this variable to `categorical`. Also, change the name of this variable to lowercase. Because `first_name` stores text data, change the data type of this variable to `string`.

```

opts = setoptions(opts,'hire_date','Type','datetime');
opts = setoptions(opts,'DEPARTMENT_NAME','Name','department_name', ...
  'Type','categorical');
opts = setoptions(opts,'first_name','Type','string');

```

```

vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)

```

```

varOpts =
  1x16 SQLVariableImportOptions array with properties:

```

```

Variable Options:

```

	(1)	(2)	(3)	(4)	(5)	(6)
Name:	'employee_id'	'first_name'	'last_name'	'email'	'phone_number'	'hire_date'
Type:	'double'	'string'	'char'	'char'	'char'	'datetime'
MissingRule:	'fill'	'fill'	'fill'	'fill'	'fill'	'fill'
FillValue:	NaN	<missing>	''	''	''	NaN

To access sub-properties of each variable, use `getoptions`

Select the three modified variables using the `SelectVariableNames` property.

```
opts.SelectedVariableNames = ["first_name", "hire_date", "department_name"];
```

Set the filter condition to import only the data for the employees hired before January 1, 2006.

```
opts.RowFilter = opts.RowFilter.hire_date < datetime(2006,01,01)
```

```
opts =
```

```
  SQLImportOptions with properties:
```

```
    ExcludeDuplicates: false
```

```
    VariableNamingRule: 'modify'
```

```
    VariableNames: {'employee_id', 'first_name', 'last_name' ... and 13 more}
```

```
    VariableTypes: {'double', 'string', 'char' ... and 13 more}
```

```
    SelectedVariableNames: {'first_name', 'hire_date', 'department_name'}
```

```
    FillValues: { NaN, <missing>, '' ... and 13 more }
```

```
    RowFilter: hire_date < 01-Jan-2006
```

```
    VariableOptions: Show all 16 VariableOptions
```

Import and display the results of the SQL query using the `fetch` function.

```
employees_data = fetch(conn,sqlquery,opts)
```

```
employees_data=4x3 table
```

first_name	hire_date	department_name
"David"	25-Jun-2005	IT
"John"	01-Oct-2004	Sales
"Karen"	05-Jan-2005	Sales
"Alberto"	10-Mar-2005	Sales

Delete the `employees` and `departments` database tables using the `execute` function.

```
execute(conn, 'DROP TABLE employees')
```

```
execute(conn, 'DROP TABLE departments')
```

Close the database connection.

```
close(conn)
```

Import Data from SQL Query as Structure

Specify the data return format and the number of imported rows for the results of an SQL query. Import data using an SQL query and the `fetch` function.

This example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the patients database table using the patient information.

```
tablename = 'patients';
sqlwrite(conn,tablename,patients)
```

Select all data from the patients database table and import five rows from the table as a structure. Use the 'DataReturnFormat' name-value pair argument to specify returning the result data as a structure. Also, use the 'MaxRows' name-value pair argument to specify five rows. Display the imported data.

```
sqlquery = ['SELECT * FROM ' tablename];
results = fetch(conn,sqlquery,'DataReturnFormat','structure', ...
    'MaxRows',5)
```

```
results = struct with fields:
    LastName: {5×1 cell}
    Gender: {5×1 cell}
    Age: [5×1 double]
    Location: {5×1 cell}
    Height: [5×1 double]
    Weight: [5×1 double]
    Smoker: [5×1 double]
    Systolic: [5×1 double]
    Diastolic: [5×1 double]
    SelfAssessedHealthStatus: {5×1 cell}
```

Delete the patients database table using the execute function.

```
sqlquery = ['DROP TABLE ' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Preserve Variable Names When Importing Data

Import product data from a Microsoft® SQL Server® database table into MATLAB® by using an ODBC connection. The table contains a variable name with a non-ASCII character. When importing data, preserve the names of all the variables.

Create an ODBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the table productTable.

```
datasource = "MSSQLServerAuth";
conn = database(datasource, "", "");
```

Check the database connection. If the Message property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Add a column to the database table `productTable`. The column name contains a non-ASCII character.

```
sqlquery = "ALTER TABLE productTable ADD tamaño varchar(30)";
execute(conn,sqlquery)
```

Import data from the database table `productTable`. The `fetch` function returns a MATLAB table that contains the product data. Display the first three rows of the data in the table.

```
sqlquery = "SELECT * FROM productTable";
data = fetch(conn,sqlquery);
head(data,3)
```

```
ans=3x6 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription  tamaño
  _____  _____  _____  _____  _____  _____
           9      1.2597e+05      1003           13      {'Victorian Doll'}  {0x0 cha
           8      2.1257e+05      1001            5      {'Train Set'   }  {0x0 cha
           7      3.8912e+05      1007           16      {'Engine Kit'   }  {0x0 cha
```

The `fetch` function converts the name of the new variable into ASCII characters.

Preserve the name of the variable that contains the non-ASCII character by specifying the `VariableNamingRule` name-value pair argument. Import the data again.

```
data = fetch(conn,sqlquery, ...
    'VariableNamingRule','preserve');
head(data,3)
```

```
ans=3x6 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription  tamaño
  _____  _____  _____  _____  _____  _____
           9      1.2597e+05      1003           13      {'Victorian Doll'}  {0x0 cha
           8      2.1257e+05      1001            5      {'Train Set'   }  {0x0 cha
           7      3.8912e+05      1007           16      {'Engine Kit'   }  {0x0 cha
```

The `fetch` function preserves the non-ASCII character in the variable name.

Close the database connection.

```
close(conn)
```

Retrieve Metadata Information About Imported Data

Retrieve metadata information when importing data from an SQL query. Import data using the `fetch` function and explore the metadata information by using dot notation.

This example uses the `outages.csv` file, which contains outage data. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MS SQL Server Auth";
conn = database(datasource, "", "");
```

Load outage information into the MATLAB® workspace.

```
outages = readtable("outages.csv");
```

Create the `outages` database table using the outage information.

```
tablename = "outages";
sqlwrite(conn, tablename, outages)
```

Import the data into the MATLAB workspace and return metadata information about the imported data.

```
sqlquery = "SELECT * FROM outages";
[results, metadata] = fetch(conn, sqlquery);
```

View the names of the variables in the imported data.

```
metadata.Properties.RowNames
```

```
ans = 6x1 cell
    {'Region'      }
    {'OutageTime'  }
    {'Loss'        }
    {'Customers'   }
    {'RestorationTime'}
    {'Cause'       }
```

View the data type of each variable in the imported data.

```
metadata.VariableType
```

```
ans = 6x1 cell
    {'char' }
    {'char' }
    {'double'}
    {'double'}
    {'char' }
    {'char' }
```

View the missing data value for each variable in the imported data.

```
metadata.FillValue
```

```
ans=6×1 cell array
{0×0 char}
{0×0 char}
{[ NaN]}
{[ NaN]}
{0×0 char}
{0×0 char}
```

View the indices of the missing data for each variable in the imported data.

```
metadata.MissingRows
```

```
ans=6×1 cell array
{ 0×1 double}
{ 0×1 double}
{604×1 double}
{328×1 double}
{ 29×1 double}
{ 0×1 double}
```

Display the first eight rows of the imported data that contain missing restoration time. `data` contains restoration time in the fifth variable. Use the numeric indices to find the rows with missing data.

```
index = metadata.MissingRows{5,1};
nullrestoration = results(index,:);
head(nullrestoration)
```

```
ans=8×6 table
      Region      OutageTime      Loss      Customers      RestorationTime
```

Region	OutageTime	Loss	Customers	RestorationTime
'SouthEast'	{'2003-01-23 00:49:00.000'}	530.14	2.1204e+05	{0×0 char}
'NorthEast'	{'2004-09-18 05:54:00.000'}	0	0	{0×0 char}
'MidWest' }	{'2002-04-20 16:46:00.000'}	23141	NaN	{0×0 char}
'NorthEast'	{'2004-09-16 19:42:00.000'}	4718	NaN	{0×0 char}
'SouthEast'	{'2005-09-14 15:45:00.000'}	1839.2	3.4144e+05	{0×0 char}
'SouthEast'	{'2004-08-17 17:34:00.000'}	624.1	1.7879e+05	{0×0 char}
'SouthEast'	{'2006-01-28 23:13:00.000'}	498.78	NaN	{0×0 char}
'West' }	{'2003-06-20 18:22:00.000'}	0	0	{0×0 char}

Delete the outages database table using the `execute` function.

```
sqlstr = "DROP TABLE ";
sqlquery = strcat(sqlstr,tablename);
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Import Data Using SQL Prepared Statement

Create an SQL prepared statement to import data from a Microsoft® SQL Server® database using a JDBC database connection. Use the SELECT SQL statement for the SQL query. Import the data from the database and display the results.

Create a JDBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MSSQLServerJDBCAuth';
conn = database(datasource, '', '');
```

Create an SQL prepared statement for importing data from the SQL Server database using the JDBC database connection. The question marks in the SELECT SQL statement indicate it is an SQL prepared statement. This statement selects all data from the database table `inventoryTable` for the inventory that has an inventory date within a specified date range.

```
query = strcat("SELECT * FROM inventoryTable ", ...
    "WHERE inventoryDate > ? AND inventoryDate < ?");
pstmt = databasePreparedStatement(conn, query)
```

```
pstmt =
    SQLPreparedStatement with properties:
```

```
        SQLQuery: "SELECT * FROM inventoryTable where inventoryDate > ? AND inventoryDate < ?"
        ParameterCount: 2
        ParameterTypes: ["string"    "string"]
        ParameterValues: {[] []}
```

`pstmt` is an `SQLPreparedStatement` object with these properties:

- `SQLQuery` — SQL prepared statement query
- `ParameterCount` — Parameter count
- `ParameterTypes` — Parameter types
- `ParameterValues` — Parameter values

Bind parameter values in the SQL prepared statement. Select both parameters in the SQL prepared statement using their numeric indices. Specify the values to bind as the inventory date range between January 1, 2014, and December 31, 2014. Match the format of dates in the database. The `bindParamValues` function updates the values in the `ParameterValues` property of the `pstmt` object.

```
selection = [1 2];
values = {"2014-01-01 00:00:00.000", ...
    "2014-12-31 00:00:00.000"};
pstmt = bindParamValues(pstmt, selection, values)
```

```
pstmt =
    SQLPreparedStatement with properties:
```

```
        SQLQuery: "SELECT * FROM inventoryTable where inventoryDate > ? AND inventoryDate < ?"
        ParameterCount: 2
        ParameterTypes: ["string"    "string"]
        ParameterValues: [{"2014-01-01 00:00:00.000"} ["2014-12-31 00:00:00.000"]}
```

Import data from the database using the `fetch` function and bound parameter values. The results contain four rows of data that represent all inventory with an inventory date between January 1, 2014 and December 31, 2014.

```
results = fetch(conn,pstmt)
```

```
results=4x4 table
  productNumber  Quantity  Price  inventoryDate
-----
           1         1700   14.5  {'2014-09-23 09:38:34'}
           2         1200    9      {'2014-07-08 22:50:45'}
           3          356   17      {'2014-05-14 07:14:28'}
           7         6034   16      {'2014-08-06 08:38:00'}
```

Close the SQL prepared statement and database connection.

```
close(pstmt)
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar. The SQL statement can be any valid SQL statement, including nested queries. The SQL statement can be a stored procedure, such as `{call sp_name (parm1,parm2,...)}`. For stored procedures that return one or more result sets, use the `fetch` function. For procedures that return output arguments, use `runstoredprocedure`.

For information about the SQL query language, see the SQL Tutorial.

Data Types: `char` | `string`

opts — Database import options

SQLImportOptions object

Database import options, specified as an `SQLImportOptions` object.

pstmt — SQL prepared statement

SQLPreparedStatement object

SQL prepared statement, specified as an `SQLPreparedStatement` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `results = fetch(conn,sqlquery,"MaxRows",50,"DataReturnFormat","structure")` imports 50 rows of data as a structure.

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of 'MaxRows' and a positive numeric scalar. By default, the `fetch` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: `'MaxRows',10`

Data Types: `double`

DataReturnFormat — Data return format

`"table"` (default) | `"cellarray"` | `"numeric"` | `"structure"`

Data return format, specified as the comma-separated pair consisting of "DataReturnFormat" and one of these values:

- `"table"`
- `"cellarray"`
- `"numeric"`
- `"structure"`

Use the "DataReturnFormat" name-value pair argument to specify the data type of the result data `results`. To specify integer classes for numeric data, use the `opts` input argument.

You can specify these values using character vectors or string scalars.

Example: `"DataReturnFormat","cellarray"` imports data as a cell array.

VariableNamingRule — Variable naming rule

`"modify"` (default) | `"preserve"`

Variable naming rule, specified as the comma-separated pair consisting of 'VariableNamingRule' and one of these values:

- `"modify"` — Remove non-ASCII characters from variable names when the `fetch` function imports data.
- `"preserve"` — Preserve most variable names when the `fetch` function imports data. For details, see the Limitations section.

Example: `'VariableNamingRule','modify'`

Data Types: `string`

RowFilter — Row filter condition

`<unconstrained>` (default) | `matlab.io.RowFilter` object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; fetch(conn,sqlquery,"RowFilter",rf)`

Output Arguments

results — Result data

table (default) | cell array | structure | numeric matrix

Result data, returned as a table, cell array, structure, or numeric matrix. The result data contains all rows of data from the executed SQL statement by default.

Use the "MaxRows" name-value pair argument to specify the number of rows of data to import. Use the "DataReturnFormat" name-value pair argument to specify the data type of the result data.

When the executed SQL statement does not return any rows, the result data is an empty table.

metadata — Metadata information

table

Metadata information, returned as a table with these variables.

Variable Name	Variable Description	Variable Data Type
VariableType	Data type of each variable in the imported data	Cell array of character vectors
FillValue	Value of missing data for each variable in the imported data	Cell array of missing data values
MissingRows	Indices for each occurrence of missing data in each variable of the imported data	Cell array of numeric indices

By default, the `fetch` function imports text data as a character vector and numeric data as a double. `FillValue` is an empty character array (for text data) or `NaN` (for numeric data) by default. To change the missing data value to another value, use the `SQLImportOptions` object.

The `RowNames` property of the metadata table contains the names of the variables in the imported data.

Limitations

The name-value argument `MaxRows` has these limitations:

- If you are using Microsoft Access, the native ODBC interface is not supported.
- Not all database drivers support setting the maximum number of rows before query execution. For an unsupported driver, modify your SQL query to limit the maximum number of rows to return. The SQL syntax varies with the driver. For details, consult the driver documentation.

The name-value argument `VariableNamingRule` has these limitations:

- The `fetch` function returns an error if you specify the `VariableNamingRule` name-value argument and set the `DataReturnFormat` name-value argument to "cellarray", "structure", or "numeric".
- The `fetch` function returns a warning if you set the `VariableNamingRule` property of the `SQLImportOptions` object to "preserve" and set the `DataReturnFormat` name-value argument to "structure".

- The `fetch` function returns an error if you use the `VariableNamingRule` name-value argument with the `SQLImportOptions` object `opts`.
- If you set the `VariableNamingRule` name-value argument to the value "modify":
 - These variable names are reserved identifiers for the `table` data type: `Properties`, `RowNames`, and `VariableNames`.
 - The length of each variable name must be less than the number returned by `namelengthmax`.

The name-value argument `RowFilter` has this limitation:

- The `fetch` function returns an error if you specify the `RowFilter` name-value argument with the `SQLImportOptions` object `opts`. It is ambiguous which of the `RowFilter` object to use in this case, especially if the filter conditions are different.

Tips

- The order of records in your database does not remain constant. Sort data using the SQL `ORDER BY` command in your `sqlquery` statement.
- For Microsoft Excel, tables in `sqlquery` are Excel worksheets. By default, some worksheet names include a \$ symbol. To select data from a worksheet with this name format, use an SQL statement of the form `SELECT * FROM "Sheet1$" (or 'Sheet1$')`.
- Before you modify database tables, ensure that the database is not open for editing. If you try to edit the database while it is open, you receive this MATLAB error:

```
[Vendor][ODBC Driver] The database engine could not lock
table 'TableName' because it is already in use by
another person or process.
```

- The PostgreSQL database management system supports multidimensional fields, but SQL `SELECT` statements fail when retrieving these fields unless you specify an index.
- Some databases require that you include a symbol, such as #, before and after a date in a query, as follows:


```
execute(conn, 'SELECT * FROM mydb WHERE mydate > #03/05/2005#')
```
- Executing the `fetch` function with the `opts` input argument and the `DataReturnFormat` name-value argument set to the "numeric" value has no effect. A corresponding warning message appears in the Command Window.

Alternative Functionality

App

The `fetch` function imports data using the command line. To import data interactively, use the **Database Explorer** app.

Version History

Introduced in R2006b

R2023a: Selectively import rows of data based on filter condition

You can use the `RowFilter` name-value argument to selectively import rows of data from a database table.

R2018b: fetch function returns results as a table

Behavior changed in R2018b

The `fetch` function returns results as a table instead of a cell array, by default. In prior releases, when the `fetch` function found no data to import, it returned a cell array containing the character vector `'No Data'`. Now, when the function finds no data to import, it returns an empty table.

R2018b: fetch function ignores 'DataReturnFormat', 'NullNumberRead', and 'NullStringRead' database preferences

Behavior changed in R2018b

The `fetch` function ignores these database preferences:

- `'DataReturnFormat'`
- `'NullNumberRead'`
- `'NullStringRead'`

You can set the data type of the imported data by using the `'DataReturnFormat'` name-value pair argument of the `fetch` function. For more customization of data types and fill values for missing data in the imported data, use the `SQLImportOptions` object.

See Also

Functions

`close` | `database` | `databaseImportOptions` | `setoptions` | `getoptions` | `reset` | `execute` | `databasePreparedStatement` | `bindParamValues` | `close`

Topics

“Import Data from Database Table Using `sqlread` Function” on page 5-58

“Retrieve Image Data Types” on page 5-48

“Data Import Memory Management” on page 5-20

“Customize Options for Importing Data from Database into MATLAB” on page 5-67

“Importing Data Common Errors” on page 3-3

“Import Data Using SQL Prepared Statement with Multiple Parameter Values” on page 5-75

“SQL Prepared Statement Error Messages” on page 3-15

External Websites

SQL Tutorial

get

Package: database.odbc

(To be removed) Retrieve object properties

Note The `get` function will be removed in a future release. There is no replacement for this functionality. To import data, use the `fetch` function. For details, see “Compatibility Considerations”.

Syntax

```
s = get(object)
v = get(object,property)
```

Description

`s = get(object)` returns the structure `s`, which contains the object and its corresponding properties.

`v = get(object,property)` returns the value `v` of property for the object.

Examples

Get cursor Object Properties

Retrieve the properties of a cursor object.

Establish an ODBC database connection to a MySQL database with the user name `username` and password `pwd`.

```
conn = database('MySQL','username','pwd');
```

Create a cursor object.

```
curs = exec(conn,'SELECT * FROM inventoryTable');
```

Retrieve the properties of `curs` and assign them as fields in the structure `v`.

```
v = get(curs)
```

```
v =
```

```
struct with fields:
    Data: 0
    RowLimit: 0
    SQLQuery: 'SELECT * FROM inventoryTable'
    Message: []
    Type: 'ODBCursor Object'
    Statement: [1x1 database.internal.ODBCStatementHandle]
    Scrollable: 0
    Position: []
```

Display the SQL query in the cursor object.

```
v.SQLQuery
ans =
    'SELECT * FROM inventoryTable'
```

Close the cursor object and database connection.

```
close(curs)
close(conn)
```

Input Arguments

object — Database Toolbox object

cursor object

Database Toolbox object, specified as a cursor object.

property — Property of Database Toolbox object

character vector | string scalar

Property of the Database Toolbox object, specified as a character vector or string scalar.

The following table shows available property names and returned values.

cursor Object Property	Description
'Data'	Data in the cursor object data element (the query results).
'RowLimit'	Maximum number of rows.
'SQLQuery'	SQL statement for a cursor object, as specified by <code>exec</code> .
'Message'	Error message returned from <code>exec</code> or <code>fetch</code> .
'Type'	Object type, specifically 'Database Cursor Object'.
'Statement'	Handle to Java statement object.
'Scrollable'	Logical value to identify the cursor object as scrollable or basic. This property is set to 1 for a scrollable cursor and 0 otherwise. This property is hidden and read-only.
'Position'	Current position of the cursor in the data set. This property is available only for a scrollable cursor. This property behaves differently for native ODBC, JDBC, and other database drivers. This property is read-only.

Data Types: char | string

Output Arguments

s — Object properties

structure

Object properties, returned as a structure that contains the object and its corresponding properties.

v — Object property value

character vector | numeric scalar | cell array | object

Object property value, returned as a character vector, numeric scalar, cell array, or object.

Version History

Introduced before R2006a

R2021a: get function will be removed

Warns starting in R2021a

The `get` function will be removed in a future release. Use the `fetch` function to import data. Some differences between the workflows might require updates to your code.

Update Code

Use the `fetch` function with the `connection` object to import data from a database in one step.

In prior releases, you wrote multiple lines of code to create the `cursor` object, retrieve object properties, and import data. For example:

```
curs = exec(conn,sqlquery);  
curs = fetch(curs);  
s = get(curs);  
results = curs.Data;  
close(curs)
```

Now you can import data in one step using the `fetch` function.

```
results = fetch(conn,sqlquery);
```

There is no replacement functionality for the `get` function.

See Also

[getdatasources](#) | [fetch](#) | [database](#) | [sqlfind](#)

Topics

“Retrieve Database Metadata” on page 5-64

External Websites

SQL Tutorial

getdatasources

(To be removed) Return names of ODBC and JDBC data sources

Note The `getdatasources` function will be removed in a future release. Use the `listDataSources` function instead.

Syntax

`d = getdatasources`

Description

`d = getdatasources` returns the names of valid ODBC and JDBC data sources on the system.

Examples

Retrieve Data Source Names

Connect to a database using its data source name.

Retrieve all ODBC and JDBC data source names on the system.

```
d = getdatasources
```

```
d =
```

```
1×11 cell array
```

```
Columns 1 through 3
```

```
    'Excel Files'    'MS Access Database'    'MS SQL Server'
```

```
...
```

`d` is a cell array of character vectors. Each character vector is a data source name that is defined on the system.

Use the data source name in the `database` function to connect to a database at the command line. Or, in the Database Explorer app, click **New Query** and then select the data source name from the **Data Source** list.

Output Arguments

d — Data sources

cell array of character vectors

Data sources, returned as a cell array of character vectors.

`d` is empty when the `ODBC.INI` file is valid, but no defined data sources exist.

For ODBC data sources, the `getdatasources` function retrieves data source names from the `ODBC.INI` file located in the folder returned by running:

```
myODBCdir = getenv('WINDIR')
```

The function also retrieves the names of data sources that are in the system registry but not in the `ODBC.INI` file.

For JDBC data sources, the `getdatasources` function retrieves data source names that you define using the JDBC Data Source Configuration dialog box in the Database Explorer app.

Version History

Introduced before R2006a

See Also

Functions

database | `listDataSources`

Apps

Database Explorer

Topics

“Connect to Database” on page 2-121

“Configure Driver and Data Source” on page 2-14

insert

Package: database.odbcc

(To be removed) Add MATLAB data to database tables

Note The `insert` function will be removed in a future release. Use the `sqlwrite` function instead. For details, see “Compatibility Considerations”.

Syntax

```
insert(conn,tablename,colnames,data)
```

Description

`insert(conn,tablename,colnames,data)` exports data from the MATLAB workspace and inserts it into an existing database table using the database connection `conn`. You can specify the database table name and column names, and specify the data for insertion into the database.

If `conn` is a JDBC database connection, then the `insert` function has the same functionality as the `fastinsert` function.

Examples

Insert Table Record Using Native ODBC

Create an ODBC database connection to the Microsoft Access database. This code assumes that you are connecting to a data source named `dbdemo` with `admin` as the user name and password.

```
conn = database('dbdemo','admin','admin');
```

This database contains the table `producttable` with these columns:

- `productnumber`
- `stocknumber`
- `suppliernumber`
- `unitcost`
- `productdescription`

Select and display the data from the `producttable` table. The cursor object contains the executed query. Import the data from the executed query using the `fetch` function.

```
curs = exec(conn,'SELECT * FROM producttable');  
curs = fetch(curs);  
curs.Data
```

```
ans =
```

```
productnumber stocknumber suppliernumber unitcost productdescription
```



```

-----
 9      125970      1003      13      'Victorian Doll'
 8      212569      1001      5       'Train Set'
 7      389123      1007      16      'Engine Kit'
 2      400314      1002      9       'Painting Set'
 4      400339      1008      21      'Space Cruiser'
 1      400345      1001      14      'Building Blocks'
 5      400455      1005      3       'Tin Soldier'
 6      400876      1004      8       'Sail Boat'
 3      400999      1009      17      'Slinky'
10     888652      1006      24      'Teddy Bear'

```

Store the column names of `producttable` in a cell array.

```
colnames = {'productnumber','stocknumber','suppliernumber',...
            'unitcost','productdescription'};
```

Store data for insertion in the cell array `data` that contains these values:

- `productnumber` equal to 11
- `stocknumber` equal to 400565
- `suppliernumber` equal to 1010
- `unitcost` equal to \$10
- `productdescription` equal to 'Rubik's Cube'

Then, convert the cell array to the table `data_table`.

```
data = {11,400565,1010,10,'Rubik's Cube'};
data_table = cell2table(data,'VariableNames',colnames)
```

```
data_table =
```

```

productnumber  stocknumber  suppliernumber  unitcost  productdescription
-----
 11            400565        1010           10        'Rubik's Cube'

```

Insert the table data into `producttable`.

```
tablename = 'producttable';
insert(conn,tablename,colnames,data_table)
```

Display the data from `producttable` again.

```
curs = exec(conn,'SELECT * FROM producttable');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```

productnumber  stocknumber  suppliernumber  unitcost  productdescription
-----
 9            125970        1003           13        'Victorian Doll'
 8            212569        1001            5        'Train Set'
 7            389123        1007           16        'Engine Kit'
 2            400314        1002            9        'Painting Set'
 4            400339        1008           21        'Space Cruiser'
 1            400345        1001           14        'Building Blocks'
 5            400455        1005            3        'Tin Soldier'
 6            400876        1004            8        'Sail Boat'
 3            400999        1009           17        'Slinky'
10            888652        1006           24        'Teddy Bear'
11            400565        1010           10        'Rubik's Cube'

```

A new row appears in `producttable` with the data from `data_table`.

After you finish working with the cursor object, close it.

`close(curs)`

Close the database connection.

`close(conn)`

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: `string` | `char`

colnames — Database table column names

cell array of character vectors | string array

Database table column names, specified as a cell array of one or more character vectors or a string array to denote the columns in the existing database table `tablename`.

Example: {'col1', 'col2', 'col3'}

Data Types: `cell` | `string`

data — Insert data

cell array | numeric matrix | table | dataset | structure

Insert data, specified as a cell array, numeric matrix, table, dataset array, or structure. These values depend on the type of database connection.

For a `connection` object, you do not specify the type of data that you are exporting. The `insert` function exports the data in its current MATLAB format. If `data` is a structure, then field names in the structure must match `colnames`. If `data` is a table or a dataset array, then the variable names in the table or dataset array must match `colnames`. If `data` is a structure, table, or dataset array, then specify each field or variable as a:

- Cell array
- Double vector of size `m-by-1`, where `m` is the number of rows to insert

For a `sqlite` object, the dataset array is not supported. Only `double`, `int64`, and `char` data types are supported.

Alternative Functionality

To export MATLAB data into a database, you can use the `datainsert` and `fastinsert` functions. For maximum performance, use `datainsert`.

Version History

Introduced before R2006a

R2022a: insert function will be removed

Warns starting in R2022a

The `insert` function will be removed in a future release. Use the `sqlwrite` function instead. Some differences between the workflows require updates to your code.

Update Code

In prior releases, you specified a cell array when exporting data from the MATLAB workspace into a database. For example:

```
colnames = {'month', 'salestotal', 'revenue'};  
data = {'March', 50, 2000};  
tablename = 'yearlysales';  
insert(conn, tablename, colnames, data)
```

Now the `sqlwrite` function requires you to specify the data to export as a table.

```
colnames = {'month', 'salestotal', 'revenue'};  
d = {'March', 50, 2000};  
data = cell2table(d, 'VariableNames', colnames);  
tablename = 'yearlysales';  
sqlwrite(conn, tablename, data)
```

See Also

`sqlwrite` | `database` | `commit` | `rollback` | `close`

Topics

“Insert Data into Database Table” on page 5-61

“Export Data Using Bulk Insert” on page 5-13

“Roll Back Data After Updating Record” on page 5-9

“Data Type Support” on page 1-3

External Websites

SQL Tutorial

insert

Add MATLAB data to SQLite database table

Note The `insert` function will be removed in a future release. Use the `sqlwrite` function instead. For details, see “Compatibility Considerations”.

Syntax

```
insert(conn,tablename,colnames,data)
```

Description

`insert(conn,tablename,colnames,data)` exports data from the MATLAB workspace and inserts it into an existing database table by using the SQLite database connection `conn`. You can specify the database table name and column names, and specify the data to be inserted into the database.

Examples

Insert Table Record Using MATLAB® Interface to SQLite

Create a table in a new SQLite database file, and then insert a new row of data into the table.

Create the SQLite connection `conn` to the new SQLite database file `tutorial.db`. Specify the file name in the current folder.

```
dbfile = fullfile(pwd,'tutorial.db');
```

```
conn = sqlite(dbfile,'create');
```

Create the table `inventoryTable` using `exec`.

```
createInventoryTable = ['create table inventoryTable ' ...  
    '(productNumber NUMERIC, Quantity NUMERIC, ' ...  
    'Price NUMERIC, inventoryDate VARCHAR)'];
```

```
exec(conn,createInventoryTable)
```

`inventoryTable` is an empty table in `tutorial.db`.

Insert a row of data into `inventoryTable`.

```
colnames = {'productNumber','Quantity','Price','inventoryDate'};
```

```
insert(conn,'inventoryTable',colnames, ...  
    {20,150,50.00,'11/3/2015 2:24:33 AM'})
```

Close the SQLite connection.

```
close(conn)
```

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: `string` | `char`

colnames — Database table column names

cell array of character vectors | string array

Database table column names, specified as a cell array of one or more character vectors or a string array to denote the columns in the existing database table `tablename`.

Example: {'col1', 'col2', 'col3'}

Data Types: `cell` | `string`

data — Insert data

numeric matrix | structure | table | cell array

Insert data, specified as a numeric matrix, structure, table, or cell array.

Version History

Introduced in R2016a

R2022a: insert function will be removed

Not recommended starting in R2022a

The `insert` function will be removed in a future release. Use the `sqlwrite` function to import data. Some differences between the workflows might require updates to your code.

Update Code

Use the `sqlwrite` function with the `sqlite` object to export data from a SQLite database.

In prior releases, you exported data using the `insert` function. For example:

```
tablename = 'inventoryTable';
colnames = {'productNumber', 'Quantity', 'Price', 'inventoryDate'};
insert(conn, tablename, colnames, ...
       {20, 150, 50.00, '11/3/2015 2:24:33 AM'})
```

Now you can export data using the `sqlwrite` function.

```
tablename = "productTable";
data = table(30, 500000, 1000, 25, "Rubik's Cube", ...
```

```
    'VariableNames',["productNumber" "stockNumber" ...  
    "supplierNumber" "unitCost" "productDescription"]);  
sqlwrite(conn,tablename,data)
```

See Also

close | sqlite | exec | fetch

Topics

“Import Data Using MATLAB Interface to SQLite” on page 5-32

“Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5

External Websites

SQL Tutorial

listDataSources

List all available data sources

Syntax

```
list = listDataSources()
```

Description

`list = listDataSources()` returns a list of all available data sources on the system. Specify a data source name using the `database` function to connect to a database at the command line. Or, in the Database Explorer app, click **Connect** in the **Connections** section, and then select a data source name from the list.

Examples

Retrieve All Data Sources

Retrieve a list of all data sources on the system.

```
list = listDataSources
```

```
list=2x3 table
      Name                DriverType          Vendor
-----
"MSSQLServerAuth"       "ODBC"             <missing>
"MSSQLServerJDBCAuth"   "JDBC"             "Microsoft SQL Server"
```

`list` is a table with three variables for the data source name, driver type, and database vendor. The variable for the database vendor contains values for JDBC and native interfaces only.

Each row represents an existing data source on the system.

Output Arguments

list — List of data sources

table

List of data sources, returned as a table.

For ODBC interfaces, the `listDataSources` function retrieves data sources that you define using the ODBC Data Source Administrator dialog box, which you can open by using the `configureODBCDataSource` function.

For JDBC and native interfaces, the `listDataSources` function retrieves data sources that you define using the JDBC Data Source Configuration dialog box or the `databaseConnectionOptions` function at the command line.

Version History

Introduced in R2020b

See Also

Functions

database | configureODBCDataSource | databaseConnectionOptions

Apps

Database Explorer

Topics

“Connect to Database” on page 2-121

“Configure Driver and Data Source” on page 2-14

rollback

Package: database.odbc

Undo database changes

Syntax

```
rollback(conn)
```

Description

`rollback(conn)` reverses changes made to a database using `datainsert`, `fastinsert`, `insert`, or `update` via the database connection `conn`. The `rollback` function reverses all changes made since the last `COMMIT` or `ROLLBACK` operation. To use `rollback`, the `AutoCommit` flag for `conn` must be `off`.

Note If the database engine is not InnoDB, `rollback` does not roll back data in MySQL databases.

Examples

- 1 Ensure that the `AutoCommit` flag for connection `conn` is `off` by running:

```
conn.AutoCommit
ans =
    'off'
```

- 2 Insert data contained in `exdata` into the columns `DEPTNO`, `DNAME`, and `LOC`, in the table `DEPT`, for the data source `conn`.

```
datainsert(conn, 'DEPT', ...
{'DEPTNO'; 'DNAME'; 'LOC'}, exdata)
```

- 3 Roll back the data `exdata` that you inserted into the database by running:

```
rollback(conn)
```

The database contains the original data present before running `datainsert`.

Tips

For ODBC connections, you can use the `rollback` function with the native ODBC interface. For details, see `database`.

Version History

Introduced before R2006a

See Also

`commit` | `datainsert` | `insert` | `update` | `get` | `database`

Topics

“Roll Back Data After Updating Record” on page 5-9

“Insert Data into Database Table” on page 5-61

“Replace Existing Data in Database” on page 5-12

rollback

Undo changes to SQLite database file

Syntax

```
rollback(conn)
```

Description

`rollback(conn)` reverses changes that you made to a database using functions such as `sqlwrite` with the MATLAB interface to SQLite. The `rollback` function reverses all changes made after the last `COMMIT` or `ROLLBACK` operation. To use this function, you must set the `AutoCommit` property of the `sqlite` object to `off`.

Examples

Reverse Changes Made to SQLite Database

Use the MATLAB® interface to SQLite to insert product data from MATLAB into a new table in an SQLite database. Then, reverse the changes made to the database.

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Allow manual committing of changes to the database by setting the `AutoCommit` property to `off`.

```
conn.AutoCommit = "off";
```

Create a MATLAB table that contains data for two products. The data is stored in the `productTable` and `suppliers` tables.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"],'VariableNames',["productNumber" ...
    "stockNumber" "supplierNumber" "unitCost" "productDescription"]);
```

Insert the product data into a new table named `toyTable`.

```
tablename = "toyTable";
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
    productNumber    stockNumber    supplierNumber    unitCost    productDescription
```

```
      30          5e+05          1000          25          "Rubik's Cube"  
      40          6e+05          2000          30          "Doll House"
```

Reverse the changes made to the database.

```
rollback(conn)
```

Import and display the contents of the database table again. The results are empty.

```
rows = sqlread(conn,tablename)
```

```
rows =
```

```
  0x5 empty table
```

Delete the new table to maintain the dataset.

```
sqlquery = "DROP TABLE toyTable";  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

Version History

Introduced in R2022a

See Also

Objects

`sqlite`

Functions

`execute` | `fetch` | `sqlwrite` | `close`

Topics

“Insert Data into SQLite Database Table” on page 5-36

“Create Table and Add Column in SQLite Database” on page 5-38

“Delete Data from SQLite Database” on page 5-39

“Roll Back Data in SQLite Database” on page 5-41

executeSQLScript

Package: database.odbc

Execute SQL script on database

Syntax

```
results = executeSQLScript(conn,scriptfile)
results = executeSQLScript(conn,scriptfile,Name,Value)
```

Description

`results = executeSQLScript(conn,scriptfile)` uses the database connection `conn` to return a structure array that contains results as a table (by default) for each executed SQL SELECT statement in the SQL script file. For any non-SELECT SQL statements, the corresponding table is empty. The `executeSQLScript` function executes all SQL statements in the SQL script file.

`results = executeSQLScript(conn,scriptfile,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'DataReturnFormat','cellarray'` stores the results of an executed SQL statement as a cell array. The results are stored in the `Data` field of the structure array.

Examples

Execute SQL Script

Connect to a Microsoft® SQL Server® database. Then, run two SQL SELECT statements from the SQL script file `compare_sales.sql`, import the results, and perform simple sales data analysis. The file contains two SQL queries in order. The first SQL query retrieves sales of products from US suppliers and the second SQL query retrieves sales of products from foreign suppliers.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Execute the SQL script. The SQL script has two SQL queries that retrieve sales data from US and foreign suppliers, respectively.

```
scriptfile = 'compare_sales.sql';
results = executeSQLScript(conn,scriptfile)
```

```
results = 1x2 struct array with fields:
  SQLQuery
  Data
  Message
```

The `executeSQLScript` function returns a structure array that contains two tables in the `Data` field. The first table contains the results of executing the first SQL query in the SQL script file. The second table contains the results of executing the second SQL query.

Display the first eight rows of imported data for the second SQL query in the SQL script file. The data shows sales results from foreign suppliers.

```
data = head(results(2).Data)
```

```
data=8x6 table
  productDescription      supplierName      city      Jan_Sales      Feb_Sales
  _____      _____      _____      _____      _____
  'Victorian Doll'      'Wacky Widgets'      'Adelaide'      1400      1100
  'Painting Set'      'Terrific Toys'      'London'      3000      2400
  'Sail Boat'      'Incredible Machines'      'Dublin'      3000      2400
  'Slinky'      'Doll's Galore'      'London'      3000      1500
  'Convertible'      'Incredible Machines'      'Dublin'      6000      3100
  'Hugsy'      'The Great Teddy Bear Company'      'Belfast'      1800      9700
  'Pancakes'      'Aunt Jemimas'      'New York'      3100      9400
  'Shawl'      'Indian Export'      'Mumbai'      235      1800
```

Retrieve the variable names in the table.

```
names = data.Properties.VariableNames
```

```
names = 1x6 cell array
  {'productDescription'}      {'supplierName'}      {'city'}      {'Jan_Sales'}      {'Feb_Sales'}
```

Determine the highest sales amount in January.

```
max(data.Jan_Sales)
```

```
ans = 6000
```

Close the database connection.

```
close(conn)
```

Execute SQL Script and Return Results as Structures

Connect to a Microsoft® SQL Server® database. Then, run two SQL `SELECT` statements from the SQL script file `compare_sales.sql`. Import the results from the SQL queries as structures and perform simple sales data analysis. The file contains two SQL queries in order. The first SQL query retrieves sales of products from US suppliers and the second SQL query retrieves sales of products from foreign suppliers.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the Message property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
    []
```

Execute the SQL script. The SQL script has two SQL queries that retrieve sales data from US and foreign suppliers, respectively. Specify `structure` as the data return format for importing the query results.

```
scriptfile = 'compare_sales.sql';
results = executeSQLScript(conn,scriptfile, ...
    'DataReturnFormat','structure')
```

```
results = 1x2 struct array with fields:
    SQLQuery
    Data
    Message
```

The `executeSQLScript` function returns a structure array that contains two structures in the `Data` field. The first structure contains the results of executing the first SQL query in the SQL script file. The second structure contains the results of executing the second SQL query.

Display the imported data for the second SQL query in the SQL script file. The data contains sales results from foreign suppliers.

```
data = results(2).Data
```

```
data = struct with fields:
    productDescription: {9x1 cell}
    supplierName: {9x1 cell}
    city: {9x1 cell}
    Jan_Sales: [9x1 double]
    Feb_Sales: [9x1 double]
    Mar_Sales: [9x1 double]
```

Determine the highest sales amount in January.

```
max(data.Jan_Sales)
```

```
ans = 6000
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

scriptfile — Name of SQL script file

character vector | string scalar

Name of SQL script file that contains one or more SQL statements to run, specified as a character vector or string scalar. The file must be a text file and can contain comments in addition to SQL queries. Start single-line comments with `--`. Enclose multiline comments in `/*...*/`.

The SQL script file can contain one or more SQL statements terminated by either a semicolon or the keyword `GO`. The following is an example of two SQL `SELECT` statements.

```
SELECT productDescription, supplierName
FROM suppliers A, productTable B
WHERE A.SupplierNumber = B.SupplierNumber;
```

```
SELECT supplierName, Country
FROM suppliers;
```

Example: `'C:\work\sql_file.sql'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `results = executeSQLScript(conn,scriptfile,'DataReturnFormat','numeric','ErrorHandling','store')` returns query results as a numeric matrix in the `Data` field of the results structure array and stores any error message in the `Message` field of results.

DataReturnFormat — Data return format

'table' (default) | 'cellarray' | 'numeric' | 'structure'

Data return format, specified as the comma-separated pair consisting of `'DataReturnFormat'` and one of these values:

- 'table'
- 'cellarray'
- 'numeric'
- 'structure'

You can specify these values using character vectors or string scalars.

The `'DataReturnFormat'` name-value pair argument specifies the data type of the `Data` field in the `results` structure array.

Example: `'DataReturnFormat','structure'` returns a structure array that contains query results stored in structures.

ErrorHandling — Error handling

`'report'` (default) | `'store'`

Error handling, specified as the comma-separated pair consisting of `'ErrorHandling'` and one of these values:

- `'report'` — When an SQL statement fails to execute, stop execution of the remaining SQL statements in the SQL script file and display an error message at the command line.
- `'store'` — When an SQL statement fails to execute, store an error message in the `Message` field of the `results` structure array.

You can specify these values using character vectors or string scalars.

Example: `'ErrorHandling','report'` displays an error message at the command line.

Output Arguments

results — Query results

structure array

Query results from executed SQL statements in the SQL script file, returned as a structure array with these fields.

Field Name	Field Data Type	Field Description
SQLQuery	character vector	Stores the SQL statement or statements executed in the SQL script file.
Data	<ul style="list-style-type: none"> • table (default) • cell array • numeric matrix • structure 	<p>Stores the results of executed SQL SELECT statements.</p> <p>The <code>'DataReturnFormat'</code> name-value pair argument specifies the data type of the <code>Data</code> field.</p> <p>For non-SELECT SQL statements, the <code>Data</code> field is an empty double, which means the executed SQL query has no results.</p>
Message	character vector	<p>Stores an error message for the respective SQL statement that fails to execute.</p> <p>The <code>Message</code> field contains an error message only if you specify the <code>'ErrorHandling'</code> name-value pair argument with the value <code>'store'</code>.</p>

The number of elements in the structure array is equal to the number of SQL statements in the SQL script file. `results(M)` contains the results from executing the *M*th SQL statement in the SQL script file. If the SQL statement returns query results, then the results are stored in `results(M).Data`.

For details about accessing structure arrays, see “Structure Arrays”.

Limitations

- Use the `executeSQLScript` function to import data into MATLAB, especially if you have long and complex SQL queries that are difficult to convert into MATLAB character vectors or string scalars. The `executeSQLScript` function does not support SQL scripts containing continuous PL/SQL blocks with `BEGIN` and `END`, such as stored procedure definitions or trigger definitions. However, `executeSQLScript` does support table definitions.
- An SQL script containing either of the following can produce unexpected results:
 - Apostrophes that are not escaped, including those in comments. For example, write the character vector 'Here's the code' as 'Here''s the code'.
 - Nested comments.
- An SQL script containing more than 25,000 characters causes the `executeSQLScript` function to return an error.

Version History

Introduced in R2019a

See Also

`database` | `close`

Topics

“Import Data from Database Table Using `sqlread` Function” on page 5-58

“Configure Driver and Data Source” on page 2-14

“Generate SQL Query and MATLAB Script” on page 4-20

“Data Import Using Database Explorer App or Command Line” on page 2-125

External Websites

SQL Tutorial

runsqlscript

Package: database.odbcc

(To be removed) Run SQL script on database

Note The `runsqlscript` function will be removed in a future release. Use the `executeSQLScript` function instead. For details, see “Compatibility Considerations”.

Syntax

```
results = runsqlscript(conn,scriptfile)
results = runsqlscript( ____,Name,Value)
```

Description

`results = runsqlscript(conn,scriptfile)` returns a cursor object array that contains a cursor object for each executed SQL command in the SQL script file `scriptfile` using the database connection. The `runsqlscript` function executes all SQL commands in the SQL script file.

`results = runsqlscript(____,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'RowInc',5` returns results from the executed SQL statements in the SQL script file in increments of five rows at a time.

Examples

Run SQL Script

First, connect to the Microsoft® SQL Server® database. Then, run two SQL SELECT statements from a SQL script file. Perform simple sales data analysis. Close the database connection.

To find the SQL script file, navigate to `\toolbox\database\dbdemos\compare_sales.sql` in your MATLAB® root folder. Copy and paste the path into your current working folder.

Create an ODBC database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Run the SQL script. The SQL script has two queries. When the SQL script executes, it returns two cursor objects that contain the imported data from each query in a cursor object array.

```
scriptfile = 'compare_sales.sql';
results = runsqlscript(conn,scriptfile)
```

```
results =
```

```
1x2 cursor array with properties:
```

```
Data
RowLimit
SQLQuery
Message
Type
Statement
Position
```

Display the cursor object for the second query.

```
results(2)
```

```
ans =
```

```
cursor with properties:
```

```
Data: [9x6 table]
RowLimit: 0
SQLQuery: 'select      productDescription, supplierName, city, January as Jan_Sales, February as Feb_Sales
from      sales'
Message: []
Type: 'ODBCCursor Object'
Statement: [1x1 database.internal.ODBCStatementHandle]
```

Display the imported data for the second query.

```
data = results(2).Data
```

```
data =
```

```
9x6 table
```

productDescription	supplierName	city	Jan_Sales	Feb_Sales
'Victorian Doll'	'Wacky Widgets'	'Adelaide'	1400	1100
'Painting Set'	'Terrific Toys'	'London'	3000	2400
'Sail Boat'	'Incredible Machines'	'Dublin'	3000	2400
'Slinky'	'Doll's Galore'	'London'	3000	1500
'Convertible'	'Incredible Machines'	'Dublin'	6000	3100
'Hugsy'	'The Great Teddy Bear Company'	'Belfast'	1800	9700
'Pancakes'	'Aunt Jemimas'	'New York'	3100	9400
'Shawl'	'Indian Export'	'Mumbai'	235	1800

```
'Snacks'           'Indian Export'           'Mumbai'           123           1700
```

Retrieve the column names for the second query.

```
names = columnnames(results(2))
```

```
names =
```

```
    'productDescription','supplierName','city','Jan_Sales','Feb_Sales','Mar_Sales'
```

Determine the highest sales amount in January.

```
max(data.Jan_Sales)
```

```
ans =
```

```
    6000
```

Close the cursor object array and database connection.

```
close(results)
close(conn)
```

Run SQL Script in Row Increments

First, connect to the Microsoft® SQL Server® database. Then, run two SQL SELECT statements from a SQL script file. Import data in one-row increments. Perform simple sales data analysis. Close the database connection.

To find the SQL script file, navigate to `\toolbox\database\dbdemos\compare_sales.sql` in your MATLAB® root folder. Copy and paste the path into your current working folder.

Create an ODBC database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
    []
```

Run the SQL script and specify one-row increments. The SQL script has two queries. When the SQL script executes, it returns two `cursor` objects that contain the imported data from each query in a `cursor` object array.

```
results = runsqlscript(conn, 'compare_sales.sql', 'RowInc', 1)
```

```
results =
```

```
1×2 cursor array with properties:
```

```
Data
RowLimit
SQLQuery
Message
Type
Statement
Position
```

Display the imported data for the second query.

```
results(2).Data
```

```
ans =
```

```
1×6 table
```

productDescription	supplierName	city	Jan_Sales	Feb_Sales	Mar_Sales
'Victorian Doll'	'Wacky Widgets'	'Adelaide'	1400	1100	981

Because of the one-row increment specification, only the first row of data is displayed.

Import the next row of data using the `fetch` function and display it.

```
curs = fetch(results(2), 1);
curs.Data
```

```
ans =
```

```
1×6 table
```

productDescription	supplierName	city	Jan_Sales	Feb_Sales	Mar_Sales
'Painting Set'	'Terrific Toys'	'London'	3000	2400	1800

Determine the highest sales amount among the months of January, February, and March.

```
data = curs.Data;
max([data.Jan_Sales data.Feb_Sales data.Mar_Sales])
```

```
ans =
```

```
3000
```

Close the cursor object array, cursor object, and database connection.

```
close(results)
close(curs)
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the database function.

scriptfile — SQL script file name

character vector | string scalar

SQL script file name that contains SQL statements to run, specified as a character vector or string scalar. The file must be a text file and can contain comments along with SQL queries. Start single-line comments with `--`. Enclose multiline comments in `/*...*/`.

Example: 'C:\work\sql_file.sql'

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `results = runsqlscript(conn,scriptfile,'RowInc',3,'QTimeOut',60);`

RowInc — Row limit

0 (default) | numeric scalar

Row limit indicating the number of rows to retrieve at a time, specified as the comma-separated pair consisting of 'RowInc' and a positive numeric scalar. Use this name-value pair argument when importing large amounts of data. Importing data in increments helps reduce overall retrieval time.

By default, the `runsqlscript` function imports all rows of data from the executed SQL statements. The value 0 specifies to import all rows of data.

Example: 'RowInc',5

Data Types: double

QTimeOut — Query timeout

0 (default) | numeric scalar

Query timeout in seconds, specified as the comma-separated pair consisting of 'QTimeOut' and a positive numeric scalar. By default, the `runsqlscript` function waits an unlimited number of seconds to execute SQL statements in the SQL script file. The value 0 specifies to wait an unlimited amount of time.

Example: 'QTimeOut',180

Data Types: double

Output Arguments

results — Query results

cursor object array

Query results from executing the SQL commands in the SQL script file, returned as a cursor object array. The number of elements in `results` is equal to the number of batches on page 11-284 in the file `scriptfile`.

`results(M)` contains the results from executing the `M`th batch in the SQL script. If the batch returns a result set, then it is stored in `results(M).Data`.

Limitations

- Use `runsqlscript` to import data into MATLAB, especially if you have long and complex SQL queries that are difficult to convert into MATLAB character vectors or string scalars. `runsqlscript` is not designed to handle SQL scripts containing continuous PL/SQL blocks with `BEGIN` and `END`, such as stored procedure definitions or trigger definitions. However, table definitions do work.
- An SQL script containing any of the following can produce unexpected results:
 - Apostrophes that are not escaped, including the ones in comments. For example, write the character vector 'Here's the code' as 'Here''s the code'.
 - Nested comments.
- An SQL script containing more than 25,000 characters causes `runsqlscript` to return an error.

More About

Batch

One or more SQL statements terminated by either a semicolon or the keyword `GO`; for example:

```
SELECT productDescription, supplierName  
FROM suppliers A, productTable B  
WHERE A.SupplierNumber = B.SupplierNumber;
```

```
SELECT supplierName, Country  
FROM suppliers;
```

Tips

- Any values assigned to `RowInc` or `QTimeOut` apply to all queries in the SQL script. For example, if 'RowInc' is set to 5, then all queries in the script return at most five rows in their respective query results.
- You can set preferences for the query results using the `setdbprefs` function. Preference settings apply to all queries in the SQL script. For example, if the 'DataReturnFormat' is set to numeric, all query results return as numeric matrices.

Version History

Introduced in R2012a

R2022a: runsqlscript function will be removed

Warns starting in R2022a

The runsqlscript function will be removed in a future release. Use the executeSQLScript function instead. Some differences between the workflows might require updates to your code.

Update Code

In prior releases, the output argument of the runsqlscript function was a cursor array. For example:

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
scriptfile = 'compare_sales.sql';
results = runsqlscript(conn, scriptfile)
```

results =

1×2 cursor array with properties:

```
Data
RowLimit
SQLQuery
Message
Type
Statement
Position
```

Now the executeSQLScript function returns a structure array.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
scriptfile = 'compare_sales.sql';
results = executeSQLScript(conn, scriptfile)
```

results = 1×2 struct array with fields:

```
SQLQuery
Data
Message
```

You can also change the data return format of the results in the structure array by using the 'DataReturnFormat' name-value pair argument. The executeSQLScript function ignores database preferences set by the setdbprefs function.

See Also

database | setdbprefs | fetch | close

Topics

“Import Data from Database Table Using sqlread Function” on page 5-58

“Configure Driver and Data Source” on page 2-14

“Generate SQL Query and MATLAB Script” on page 4-20

“Data Import Using Database Explorer App or Command Line” on page 2-125

runstoredprocedure

Package: database.odbc

Call stored procedure with and without input and output arguments

Syntax

```
results = runstoredprocedure(conn, sname)
results = runstoredprocedure(conn, sname, inputargs)
results = runstoredprocedure(conn, sname, inputargs, outputtypes)
```

Description

This function calls a stored procedure that has no input arguments, no output arguments, or any combination of input and output arguments. Define and instantiate this stored procedure in your database.

You can use this function if you connect to your database using a JDBC driver. For details, see “Connect to Database” on page 2-121. If you are using the native ODBC interface to connect to your database, use `execute` to call the stored procedure.

`results = runstoredprocedure(conn, sname)` calls the stored procedure `sname` using the database connection `conn`. `results` is a logical 1 if the stored procedure returns a data set. Otherwise, `results` is a logical 0.

`results = runstoredprocedure(conn, sname, inputargs)` calls the stored procedure that accepts one or more input arguments `inputargs`.

`results = runstoredprocedure(conn, sname, inputargs, outputtypes)` calls the stored procedure that returns output arguments by specifying the output argument data types `outputtypes`. `results` is a cell array that contains one or more output arguments.

Examples

Call a Stored Procedure Without Input and Output Arguments

Define a stored procedure named `create_table` that creates a table named `test_table` by executing this code. This procedure has no input or output arguments. This code assumes you are using a Microsoft SQL Server database.

```
CREATE PROCEDURE create_table
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

CREATE TABLE test_table
```

```
(
    CATEGORY_ID    INTEGER    IDENTITY PRIMARY KEY,
    CATEGORY_DESC CHAR(50)    NOT NULL
);
```

```
END
GO
```

Create a Microsoft SQL Server database connection `conn` using the JDBC driver. For details, see “Connect to Database” on page 2-121. Then, call the stored procedure `create_table` using the database connection `conn`.

```
results = runstoredprocedure(conn, 'create_table')
```

```
results =
```

```
0
```

`results` returns 0 because calling `create_table` does not return a data set.

Check your database for a new table named `test_table`.

Close the database connection `conn`.

```
close(conn)
```

Call a Stored Procedure with Input Arguments

Define a stored procedure named `insert_data` that inserts a category description into a table named `test_create` by executing this code. This procedure has one input argument `data`. This code assumes you are using a Microsoft SQL Server database.

```
CREATE PROCEDURE insert_data
    @data varchar(50)

AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    INSERT INTO test_create (CATEGORY_DESC)
    VALUES (@data)
END
GO
```

Create a Microsoft SQL Server database connection `conn` using the JDBC driver. For details, see “Connect to Database” on page 2-121. Then, call the stored procedure `insert_data` using the database connection `conn` with the category description `Apples` as the input argument.

```
inputarg = {'Apples'};
```

```
results = runstoredprocedure(conn, 'insert_data', inputarg)
```

```
results =
```

```
0
```

`results` returns 0 because calling `insert_data` does not return a data set.

The table `test_create` adds a row where the column `CATEGORY_ID` equals 1 and the column `CATEGORY_DESCRIPTION` equals Apples.

`CATEGORY_ID` is the primary key of the table `test_create`. This primary key increments automatically. `CATEGORY_ID` equals 1 when calling `insert_data` for the first time.

Close the database connection `conn`.

```
close(conn)
```

Call a Stored Procedure with Output Arguments

Define a stored procedure named `maxDecVolume` that selects the maximum sales volume in December by executing this code. This procedure has one output argument `data` and no input arguments. This code assumes you are using a Microsoft SQL Server database.

```
CREATE PROCEDURE maxDecVolume
    @data int OUTPUT
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    SELECT @data = max(December) FROM salesVolume
END
GO
```

Create a Microsoft SQL Server database connection `conn` using the JDBC driver. For details, see “Connect to Database” on page 2-121. Then, call the stored procedure using:

- Database connection `conn`
- Stored procedure `maxDecVolume`
- Empty brackets to denote no input arguments
- Numeric Java data type `outputtype`

```
outputtype = {java.sql.Types.NUMERIC};
```

```
results = runstoredprocedure(conn, 'maxDecVolume', [], outputtype)
```

```
results =
```

```
    [1x1 java.math.BigDecimal]
```

`results` returns a cell array that contains the maximum sales volume as a Java decimal data type.

Display the value in `results`.

```
results{1}
```

```
ans =
```

```
35000
```

The maximum sales volume in December is 35,000.

Close the database connection `conn`.

```
close(conn)
```

Call a Stored Procedure with Input and Output Arguments

Define a stored procedure named `getSuppCount` that counts the number of suppliers for a specified city by executing this code. This procedure has one input argument `cityName` and one output argument `suppCount`. This code assumes you are using a Microsoft SQL Server database.

```
CREATE PROCEDURE getSuppCount
    (@cityName varchar(20),
    @suppCount int OUTPUT)
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    SELECT @suppCount = count(supplierNumber)
    FROM suppliers WHERE City = @cityName;

END
GO
```

Create a Microsoft SQL Server database connection `conn` using the JDBC driver. For details, see “Connect to Database” on page 2-121. Then, call the stored procedure `getSuppCount` using the database connection `conn`. The input argument `inputarg` is a cell array containing the character vector `'New York'`. The output Java data type `outputtype` is numeric.

```
inputarg = {'New York'};
outputtype = {java.sql.Types.NUMERIC};

results = runstoredprocedure(conn, 'getSuppCount', inputarg, outputtype)

results =

    [1x1 java.math.BigDecimal]
```

`results` is a cell array that contains the supplier count as a Java decimal data type.

Display the value in `results`.

```
results{1}
```

```
ans =
```

```
6.0000
```

There are six suppliers in New York.

Close the database connection conn.

```
close(conn)
```

Call a Stored Procedure with Multiple Input and Output Arguments

Define a stored procedure named `productsWithinUnitCost` that returns the product number and description for products that have a unit cost in a specified range by executing this code. This procedure has two input arguments `minUnitCost` and `maxUnitCost`. This procedure has two output arguments `productno` and `productdesc`. This code assumes you are using a Microsoft SQL Server database.

```
CREATE PROCEDURE productsWithinUnitCost
    (@minUnitCost INT,
    @maxUnitCost INT,
    @productno INT OUTPUT,
    @productdesc VARCHAR(50) OUTPUT)
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    SELECT @productno = productNumber, @productdesc = productDescription
    FROM productTable
    WHERE unitCost > @minUnitCost AND unitCost < @maxUnitCost
END
```

```
GO
```

Create a Microsoft SQL Server database connection `conn` using the JDBC driver. For details, see “Connect to Database” on page 2-121. Then, call the stored procedure using:

- Database connection `conn`
- Stored procedure `productsWithinUnitCost`
- Input arguments `inputargs` to specify a unit cost between 19 and 21
- Output Java data types `outputtypes` to specify numeric and string data types for product number and description

```
inputargs = {19,21};
outputtypes = {java.sql.Types.NUMERIC,java.sql.Types.VARCHAR};

results = runstoredprocedure(conn,'productsWithinUnitCost',...
    inputargs,outputtypes)
```

```
results =
```

```
    [1x1 java.math.BigDecimal]
    'Snacks'
```

`results` returns a cell array that contains the product number as a Java decimal data type and the product description as a string.

Display the product number in `results`.

```
results{1}
```

```
ans =
```

```
15
```

The product with product number 15 has a unit cost between 19 and 21.

Display the product description in `results`.

```
results{2}
```

```
ans =
```

```
Snacks
```

The product with product number 15 has the product description Snacks.

Here, the narrow unit cost range returns only one product. If the unit cost range is wider, then more than one product might satisfy this condition. To return a data set with numerous products, use `exec` and `fetch` to call this stored procedure. Otherwise, `runstoredprocedure` returns only the last row in the data set.

Close the database connection `conn`.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

spname — Stored procedure name

character vector

Stored procedure name, specified as a character vector that contains the name of the stored procedure that is defined and instantiated in your database.

Data Types: `char`

inputargs — Input arguments

cell array

Input arguments, specified as a cell array of one or more values for each input argument of the stored procedure. Input arguments can be only basic data types such as double, character vector, logical, and so on.

Data Types: `cell`

outputtypes — Output types

cell array

Output types, specified as a cell array of one or more Java data types for the output arguments of the stored procedure. Some JDBC drivers do not support all `java.sql.Types`. Consult your JDBC driver

documentation to find the supported types. Match them to the data types found in your stored procedure.

Example: `{java.sql.Types.NUMERIC}`

Data Types: `cell`

Output Arguments

results — Stored procedure results

logical | cell array

Stored procedure results, returned as a logical or cell array.

`runstoredprocedure` returns a logical 1 when calling the stored procedure returns a data set. Otherwise, `runstoredprocedure` returns a logical 0. If the stored procedure returns a data set, use `exec` and `fetch` to call the stored procedure and retrieve the data set. For details, see “Call Stored Procedure That Returns Data” on page 5-17.

`runstoredprocedure` returns a cell array when you specify one or more output Java data types for the output arguments of the stored procedure. Use cell array indexing to retrieve the output argument values.

Version History

Introduced in R2006b

See Also

`database` | `exec` | `fetch` | `close`

Topics

“Call Stored Procedure That Returns Data” on page 5-17

“Connect to Database” on page 2-121

External Websites

SQL Tutorial

select

Package: database.odbcc

Execute SQL SELECT statement and import data into MATLAB

Syntax

```
data = select(conn,selectquery)
data = select(conn,selectquery,Name,Value)
[data,metadata] = select( ___ )
```

Description

`data = select(conn,selectquery)` returns imported data from the database connection `conn` for the specified SQL SELECT statement `selectquery`.

`data = select(conn,selectquery,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, 'MaxRows', 10 sets the maximum number of rows to return to 10 rows.

`[data,metadata] = select(___)` returns information about the imported data using any of the input argument combinations in the previous syntaxes. Use this information to change missing values in the imported data and view data types for each variable.

Examples

Import and Access Data Immediately

Import data from a database in one step using the `select` function. You can access data and perform immediate data analysis.

The code assumes that you have a database table `Patients` stored on a Microsoft® SQL Server® database. This table contains patient data in 10 columns and rows. The table definition is:

```
CREATE TABLE Patients(
    LastName VARCHAR(50),
    Gender VARCHAR(10),
    Age TINYINT,
    Location VARCHAR(300),
    Height SMALLINT,
    Weight SMALLINT,
    Smoker BIT,
    Systolic FLOAT,
    Diastolic NUMERIC,
    SelfAssessedHealthStatus VARCHAR(20))
```

This example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft® SQL Server® Driver 11.00.5058.

Create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Import all data from the Patients table by executing the SQL SELECT statement using the select function. data is a table that contains the imported data.

```
selectquery = 'SELECT * FROM Patients';
data = select(conn,selectquery)
```

data =

10×10 table

LastName	Gender	Age	Location	Height	Weight	Smoker
'Smith'	'Male'	38	'Country General Hospital'	-32768	176	true
'Johnson'	'Male'	43	'VA Hospital'	69	163	false
'Williams'	'Female'	38	'VA Hospital'	64	131	false
'Jones'	'Female'	0	'VA Hospital'	67	133	false
'Broen'	'Female'	49	'Country General Hospital'	64	119	false
'Davis'	'Female'	46	'St Mary's Medical Center'	68	142	false
'Miller'	'Female'	33	'VA Hospital'	64	142	true
'Wilson'	'Male'	40	'VA Hospital'	-32768	180	false
'Moore'	'Male'	28	'St Mary's Medical Center'	68	-32768	false
'Taylor'	'Female'	31	'Country General Hospital'	68	132	false

Determine the number of male patients by immediately accessing the data. Use the count function to find occurrences in the gender data of the character vector that represents a male. Determine the total number of occurrences.

```
males = count(data.Gender, 'Male');
sum(males)
```

ans =

4

Close the database connection.

```
close(conn)
```

Limit Number of Rows in Imported Data

Import a limited number of rows from a database in one step using the select function. Database Toolbox™ imports the data using MATLAB® numeric data types that correspond to data types in the database table. After importing data, you can access data and perform immediate data analysis.

The code assumes that you have a database table `Patients` stored on a Microsoft® SQL Server® database. This table contains patient data in 10 columns and rows. The table definition is:

```
CREATE TABLE Patients(
    LastName VARCHAR(50),
    Gender VARCHAR(10),
    Age TINYINT,
    Location VARCHAR(300),
    Height SMALLINT,
    Weight SMALLINT,
    Smoker BIT,
    Systolic FLOAT,
    Diastolic NUMERIC,
    SelfAssessedHealthStatus VARCHAR(20))
```

Here, connect to a Microsoft® SQL Server® Version 11.00.2100 database using the Microsoft® SQL Server® Driver 11.00.5058.

Create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Import data from the `Patients` table by executing the SQL `SELECT` statement using the `select` function. Limit the number of imported rows using the name-value pair argument `'MaxRows'`.

`data` is a table. The MATLAB® data types in the table correspond to the data types in the database. Here, `Age` has data type `uint8` that corresponds to `TINYINT` in the table definition.

`metadata` is a table that contains additional information about each variable in `data`.

- `VariableType` -- MATLAB® data type
- `MissingValue` -- NULL value representation
- `MissingRows` -- Vector of row indices that contain a missing value

```
selectquery = 'SELECT * FROM Patients';
```

```
[data,metadata] = select(conn,selectquery, 'MaxRows',5)
```

```
data =
```

```
5×10 table
```

LastName	Gender	Age	Location	Height	Weight	Smoker
'Smith'	'Male'	38	'Country General Hospital'	-32768	176	true
'Johnson'	'Male'	43	'VA Hospital'	69	163	false
'Williams'	'Female'	38	' '	64	131	false
'Jones'	'Female'	0	'VA Hospital'	67	133	false
'Broen'	'Female'	49	'Country General Hospital'	64	119	false

```
metadata =
```

10×3 table

	VariableType	MissingValue	MissingRows
LastName	'char'	''	[0×1 double]
Gender	'char'	''	[0×1 double]
Age	'uint8'	[0]	[4]
Location	'char'	''	[0×1 double]
Height	'int16'	[-32768]	[1]
Weight	'int16'	[-32768]	[0×1 double]
Smoker	'logical'	[0]	[0×1 double]
Systolic	'single'	[NaN]	[2]
Diastolic	'double'	[NaN]	[0×1 double]
SelfAssessedHealthStatus	'char'	''	[0×1 double]

Determine the number of male patients by immediately accessing the data. Use the `count` function to find occurrences in the gender data of the character vector that represents a male. Determine the total number of occurrences.

```
males = count(data.Gender, 'Male');
sum(males)
```

```
ans =
     2
```

Close the database connection.

```
close(conn)
```

View Information About Imported Data

Import data from a database in one step using the `select` function. Database Toolbox™ imports the data using MATLAB® numeric data types that correspond to data types in the database table. You can view data type information in the imported data. You can also access data and perform immediate data analysis.

The code assumes that you have a database table `Patients` stored on a Microsoft® SQL Server® database. This table contains patient data in 10 columns and rows. The table definition is:

```
CREATE TABLE Patients(
    LastName VARCHAR(50),
    Gender VARCHAR(10),
    Age TINYINT,
    Location VARCHAR(300),
    Height SMALLINT,
    Weight SMALLINT,
    Smoker BIT,
    Systolic FLOAT,
    Diastolic NUMERIC,
    SelfAssessedHealthStatus VARCHAR(20))
```

Here, connect to a Microsoft® SQL Server® Version 11.00.2100 database using the Microsoft® SQL Server® Driver 11.00.5058.

Create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Import all data from the Patients table by executing the SQL SELECT statement using the select function.

data is a table. The MATLAB® data types in the table correspond to the data types in the database. Here, Age has the MATLAB® data type uint8 that corresponds to TINYINT in the table definition.

metadata is a table that contains additional information about each variable in data.

- VariableType -- MATLAB® data type
- MissingValue -- Null value representation
- MissingRows -- Vector of row indices that contain a missing value

```
selectquery = 'SELECT * FROM Patients';
```

```
[data,metadata] = select(conn,selectquery)
```

```
data =
```

```
10×10 table
```

LastName	Gender	Age	Location	Height	Weight	Smoker
'Smith'	'Male'	38	'Country General Hospital'	-32768	176	true
'Johnson'	'Male'	43	'VA Hospital'	69	163	false
'Williams'	'Female'	38	''	64	131	false
'Jones'	'Female'	0	'VA Hospital'	67	133	false
'Broen'	'Female'	49	'Country General Hospital'	64	119	false
'Davis'	'Female'	46	'St Mary's Medical Center'	68	142	false
'Miller'	'Female'	33	'VA Hospital'	64	142	true
'Wilson'	'Male'	40	'VA Hospital'	-32768	180	false
'Moore'	'Male'	28	'St Mary's Medical Center'	68	-32768	false
'Taylor'	'Female'	31	'Country General Hospital'	68	132	false

```
metadata =
```

```
10×3 table
```

	VariableType	MissingValue	MissingRows
LastName	'char'	''	[0×1 double]
Gender	'char'	''	[0×1 double]
Age	'uint8'	[0]	[4]
Location	'char'	''	[0×1 double]
Height	'int16'	[-32768]	[2×1 double]

```

Weight          'int16'          [-32768]          [          9]
Smoker          'logical'        [          0]      [0×1 double]
Systolic        'single'         [   NaN]          [2×1 double]
Diastolic        'double'         [   NaN]          [          6]
SelfAssessedHealthStatus 'char'          ''                [0×1 double]

```

View data types of each variable in the table.

```
metadata.VariableType
```

```
ans =
```

```
10×1 cell array
```

```

'char'
'char'
'uint8'
'char'
'int16'
'int16'
'logical'
'single'
'double'
'char'

```

Determine the number of male patients by immediately accessing the data. Use the `count` function to find occurrences in the gender data of the character vector that represents a male. Determine the total number of occurrences.

```
males = count(data.Gender, 'Male');
sum(males)
```

```
ans =
```

```
4
```

Close the database connection.

```
close(conn)
```

Change Missing Values in Imported Data Using for Loop

Import data from a database in one step using the `select` function. During import, the `select` function sets default values for missing data in each row. Use the information about the imported data to change the default values.

The code assumes that you have a database table `Patients` stored on a Microsoft® SQL Server® database. This table contains patient data in 10 columns and rows. The table definition is:

```
CREATE TABLE Patients(
    LastName VARCHAR(50),
```

```

Gender VARCHAR(10),
Age TINYINT,
Location VARCHAR(300),
Height SMALLINT,
Weight SMALLINT,
Smoker BIT,
Systolic FLOAT,
Diastolic NUMERIC,
SelfAssessedHealthStatus VARCHAR(20)

```

Here, connect to a Microsoft® SQL Server® Version 11.00.2100 database using the Microsoft® SQL Server® Driver 11.00.5058.

Create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```

datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');

```

Import all data from the Patients table by executing the SQL SELECT statement using the select function.

data is a table that contains the imported data.

metadata is a table that contains additional information about each variable in data.

- VariableType -- MATLAB® data type
- MissingValue -- NULL value representation
- MissingRows -- Vector of row indices that indicate the location of missing values

```
selectquery = 'SELECT * FROM Patients';
```

```
[data,metadata] = select(conn,selectquery)
```

```
data =
```

```
10×10 table array
```

LastName	Gender	Age	Location	Height	Weight	Smoker
'Smith'	'Male'	38	'Country General Hospital'	-32768	176	true
'Johnson'	'Male'	43	'VA Hospital'	69	163	false
'Williams'	'Female'	38	''	64	131	false
'Jones'	'Female'	0	'VA Hospital'	67	133	false
'Broen'	'Female'	49	'Country General Hospital'	64	119	false
'Davis'	'Female'	46	'St Mary's Medical Center'	68	142	false
'Miller'	'Female'	33	'VA Hospital'	64	142	true
'Wilson'	'Male'	40	'VA Hospital'	-32768	180	false
'Moore'	'Male'	28	'St Mary's Medical Center'	68	-32768	false
'Taylor'	'Female'	31	'Country General Hospital'	68	132	false

```
metadata =
```

```
10×3 table array
```


	VariableType	MissingValue	MissingRows
LastName	'char'	''	[0×1 double]
Gender	'char'	''	[0×1 double]
Age	'uint8'	[0]	[4]
Location	'char'	''	[0×1 double]
Height	'int16'	[-32768]	[2×1 double]
Weight	'int16'	[-32768]	[9]
Smoker	'logical'	[0]	[0×1 double]
Systolic	'single'	[NaN]	[2×1 double]
Diastolic	'double'	[NaN]	[6]
SelfAssessedHealthStatus	'char'	''	[0×1 double]

Retrieve indices that indicate the location of missing values in the `Height` variable using the `metadata` output argument.

```
values = metadata.MissingRows{'Height'}
```

```
values =
```

```
1
8
```

Change the default value for missing data from `-32768` to `0` using a for loop. Access the imported data using the indices.

```
for i = 1:length(values)
    data.Height(values(i)) = 0;
end
```

View the imported data.

```
data.Height
```

```
ans =
```

```
10×1 int16 column vector
```

```
0
69
64
67
64
68
64
0
68
68
```

Missing values appear as `0`.

Close the database connection.

```
close(conn)
```

Change Missing Values in Imported Data Using Vector Indexing

Import data from a database in one step using the `select` function. During import, the `select` function sets default values for missing data in each row. Use the information about the imported data to change default values by indexing into the vector.

The code assumes that you have a database table `Patients` stored on a Microsoft® SQL Server® database. This table contains patient data in 10 columns and rows. The table definition is:

```
CREATE TABLE Patients(
    LastName VARCHAR(50),
    Gender VARCHAR(10),
    Age TINYINT,
    Location VARCHAR(300),
    Height SMALLINT,
    Weight SMALLINT,
    Smoker BIT,
    Systolic FLOAT,
    Diastolic NUMERIC,
    SelfAssessedHealthStatus VARCHAR(20))
```

Here, connect to a Microsoft® SQL Server® Version 11.00.2100 database using the Microsoft® SQL Server® Driver 11.00.5058.

Create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Import all data from the `Patients` table by executing the SQL `SELECT` statement using the `select` function.

`data` is a table that contains the imported data.

`metadata` is a table that contains additional information about each variable in `data`.

- `VariableType` -- MATLAB® data type
- `MissingValue` -- NULL value representation
- `MissingRows` -- Vector of row indices that indicate the location of missing values

```
selectquery = 'SELECT * FROM Patients';
```

```
[data,metadata] = select(conn,selectquery)
```

```
data =
```

```
10×10 table array
```

LastName	Gender	Age	Location	Height	Weight	Smoker
----------	--------	-----	----------	--------	--------	--------

```

'Smith'      'Male'      38      'Country General Hospital'  -32768      176      true
'Johnson'   'Male'      43      'VA Hospital'                69          163      false
'Williams'   'Female'    38      ''                            64          131      false
'Jones'      'Female'    0       'VA Hospital'                67          133      false
'Broen'      'Female'    49      'Country General Hospital'   64          119      false
'Davis'      'Female'    46      'St Mary's Medical Center'   68          142      false
'Miller'     'Female'    33      'VA Hospital'                64          142      true
'Wilson'     'Male'      40      'VA Hospital'                -32768      180      false
'Moore'      'Male'      28      'St Mary's Medical Center'   68          -32768   false
'Taylor'     'Female'    31      'Country General Hospital'   68          132      false

```

```
metadata =
```

```
10x3 table array
```

	VariableType	MissingValue	MissingRows
LastName	'char'	''	[0x1 double]
Gender	'char'	''	[0x1 double]
Age	'uint8'	[0]	[4]
Location	'char'	''	[0x1 double]
Height	'int16'	[-32768]	[2x1 double]
Weight	'int16'	[-32768]	[9]
Smoker	'logical'	[0]	[0x1 double]
Systolic	'single'	[NaN]	[2x1 double]
Diastolic	'double'	[NaN]	[6]
SelfAssessedHealthStatus	'char'	''	[0x1 double]

Retrieve indices that indicate the location of missing values in the `Height` variable using the `metadata` output argument.

```
values = metadata(5,3)
valuesindex = values.MissingRows{1}
```

```
values =
```

```
table
```

	MissingRows
Height	[2x1 double]

```
valuesindex =
```

```
1
8
```

Change the default value for missing data from `-32768` to `0` using vector indexing.

```
data.Height(valuesindex) = 0;
```

View the imported data.

```
data.Height
```

```
ans =
```

```
10x1 int16 column vector
```

```
0  
69  
64  
67  
64  
68  
64  
0  
68  
68
```

Missing values appear as 0.

Close the database connection.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

selectquery — SQL SELECT statement

character vector | string

SQL SELECT statement, specified as a character vector or string. The `select` function only executes SQL SELECT statements. To execute other SQL statements, use the `exec` function.

Example: 'SELECT * FROM inventoryTable'

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: 'MaxRows',100,'QueryTimeOut',5 returns 100 rows of data and waits 5 seconds to execute the SQL SELECT statement.

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of 'MaxRows' and a positive numeric scalar. By default, the `select` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: 'MaxRows',10

Data Types: double

QueryTimeout — SQL query timeout

positive numeric scalar

SQL query timeout, specified as the comma-separated pair consisting of 'QueryTimeout' and a positive numeric scalar. By default, the `select` function ignores the timeout value. Use this name-value pair argument to specify the number of seconds to wait for executing the SQL query `selectquery`.

Example: 'QueryTimeout',15

Output Arguments

data — Imported data

table

Imported data, returned as a table. The rows of the table correspond to the rows of data returned from the executed SQL query `selectquery`. The variable names of the table specify the columns in the SQL query.

The `select` function returns date or time data as character vectors in the table. This function returns text as character vectors or a cell array of character vectors. Strings are not supported in the table.

If no data to import exists, then `data` is an empty table.

metadata — Information about imported data

table

Information about imported data, returned as a table. The row names of `metadata` are variable names in `data`. This function stores each variable name in the `metadata` table as a cell array. `metadata` has these variable names:

- `VariableType` — Data types of each variable in `data`
- `MissingValue` — Representation of missing value for each variable in `data`
- `MissingRows` — Vector of row indices that indicate locations of missing values for each variable in `data`

This table shows how MATLAB represents NULL values in the database by default after data import.

Database Data Type	Default NULL Value
SIGNED TINYINT	-128
UNSIGNED TINYINT	0
SIGNED SMALLINT	-32768
UNSIGNED SMALLINT	0
SIGNED INT	-2147483648

Database Data Type	Default NULL Value
UNSIGNED INT	0
SIGNED BIGINT	-9223372036854775808
UNSIGNED BIGINT	0
REAL	NaN
FLOAT	NaN
DOUBLE	NaN
DECIMAL	NaN
NUMERIC	NaN
Boolean	false
Date, time, or text	' '

To change the NULL value representation in the imported data, replace the default value by looping through the imported data or using vector indexing.

Limitations

- You cannot customize missing values in the output argument `data` using the `select` function. Index into the imported data using the `metadata` output argument instead.
- The output argument `data` does not support `cell` and `struct` data types. The `select` function only supports `table`.

Alternative Functionality

Use the `exec` and `fetch` functions for full functionality when importing data. For differences between the `select` function and this alternative, see “Data Import Using Database Explorer App or Command Line” on page 2-125.

Version History

Introduced in R2017a

See Also

`exec` | `fetch` | `database` | `close` | `count`

Topics

“Data Import Using Database Explorer App or Command Line” on page 2-125

“Data Import Memory Management” on page 5-20

“Import Data from Database Table Using `sqlread` Function” on page 5-58

External Websites

SQL Tutorial

setdbprefs

(Not recommended) Set preferences for retrieval format, errors, NULLs, and more

Note The `setdbprefs` function is not recommended. For details about functionality to use instead, see “Compatibility Considerations”.

Syntax

```
setdbprefs
v = setdbprefs
setdbprefs(preference)

setdbprefs(preference,value)
setdbprefs(s)
```

Description

`setdbprefs` returns current values for database preferences.

`v = setdbprefs` returns current values to the structure `v`.

`setdbprefs(preference)` returns the current value for the specified preference.

`setdbprefs(preference,value)` sets the specified preference to `value`. After you set database preferences, they are retained across MATLAB sessions.

`setdbprefs(s)` sets preferences specified in the structure `s` to values that you specify.

Examples

Display Current Values

Display all database preferences and their current values.

```
setdbprefs
ans =
    struct with fields:
        DataReturnFormat: 'table'
        ErrorHandling: 'store'
        NullNumberRead: 'NaN'
        NullNumberWrite: 'NaN'
        NullStringRead: 'null'
        NullStringWrite: 'null'
```

Display the current value for the specified database preference.

```
setdbprefs('ErrorHandling')
```

```
ans =  
    'report'
```

Change Preference Setting

Set a database preference to a different value. Change the display of errors in MATLAB by modifying the database error handling preference.

Specify the store format for the `ErrorHandling` preference.

```
setdbprefs('ErrorHandling','store')
```

When you execute the `database` function, Database Toolbox stores any generated errors in the `Message` property of the returned connection object.

Establish the connection `conn` to a MySQL database with the user name `username` and an invalid password.

```
conn = database('MySQL','username','invalid');
```

Access the error message in the `Message` property of the connection object.

```
conn.Message
```

```
ans =  
    'ODBC Driver Error: [MySQL][ODBC 5.3(a) Driver]Access denied for user 'username'@'servername'
```

Specify the report format for the `ErrorHandling` preference.

```
setdbprefs('ErrorHandling','report')
```

Connect to the database using the invalid password again. With the `ErrorHandling` preference set to report, the error generated by running the `database` function appears immediately in the Command Window.

```
conn = database('MySQL','username','invalid')
```

```
Error using database (line 156)  
ODBC Driver Error: [MySQL][ODBC 5.3(a) Driver]Access denied for user  
'username'@'servername' (using password: YES)
```

Assign Values to Structure

Assign values for specific preferences in a structure so you can change multiple database preferences simultaneously.

Assign values for preferences to fields in the structure `s`.

```
s.ErrorHandling = 'report';  
s.NullStringRead = 'null';  
s
```



```
s =
    struct with fields:
        ErrorHandling: 'report'
        NullStringRead: 'null'
```

Set preferences using the values in `s`.

```
setdbprefs(s)
```

Run `setdbprefs` to check your database preference settings.

```
setdbprefs
```

```
ans =
    struct with fields:
        DataReturnFormat: 'table'
        ErrorHandling: 'report'
        NullNumberRead: 'NaN'
        NullNumberWrite: 'NaN'
        NullStringRead: 'null'
        NullStringWrite: 'null'
```

Return Values to Structure

Assign values for all database preferences to `s`.

```
s = setdbprefs
```

```
s =
    struct with fields:
        DataReturnFormat: 'table'
        ErrorHandling: 'report'
        NullNumberRead: 'NaN'
        NullNumberWrite: 'NaN'
        NullStringRead: 'null'
        NullStringWrite: 'null'
```

Save Preferences

Save your database preferences to the MAT-file to use them in future MATLAB sessions.

Assign the preferences to the variable `ImportData` and save them to a MAT-file `ImportDataPrefs` in your current folder.

```
ImportData = setdbprefs;
save ImportDataPrefs.mat ImportData
```

Load the data and restore the preferences.

```
load ImportDataPrefs.mat
setdbprefs(ImportData)
```

Input Arguments

preference — Database preference

character vector | cell array

Database preference, specified as a character vector or cell array. To set multiple database preferences, enter the preference values in a cell array of character vectors. Then, match the order with the corresponding values in the `value` argument.

You can specify database preferences for error handling and importing NULL strings from a database into MATLAB.

- 'ErrorHandling' — Specify how to handle errors when importing data. Set this parameter before you execute the `database` function. To specify displaying errors in the Command Window, enter `setdbprefs('ErrorHandling','report')`. Otherwise, you can access the error message in the `Message` property of the connection object.
- NULL data — Specify how to import NULL strings into the MATLAB workspace. To import NULL strings as the character vector 'null', enter `setdbprefs('NullStringRead','null')`. Set this parameter before running `fetch`.

Example: 'ErrorHandling'

Example: {'ErrorHandling';'NullStringRead'}

Data Types: char

value — Database preference value

character vector | cell array

Database preference value, specified as a character vector or cell array. To set multiple database preferences, enter the preference values in a cell array of character vectors. Then, match the order with the corresponding preferences in the `preference` argument.

Example: 'NaN'

Example: {'numeric';'NaN'}

Data Types: char

s — Database preferences

structure

Database preferences, specified as a structure that includes all the preferences you specify.

Data Types: struct

Output Arguments

v — Database preferences

structure

Database preferences, returned as a structure containing database preference settings and values.

Alternative Functionality

For a visual way to set database preferences, click **Preferences** in the **Environment** section of the MATLAB toolstrip, and then click **Database Toolbox**. Enter values for each database preference.

Version History

Introduced before R2006a

R2019a: setdbprefs function is not recommended

Not recommended starting in R2019a

The `setdbprefs` function is not recommended. Use the following replacement functionality to specify the data return format, error handling, and missing data. Some differences between the workflows might require updates to your code.

- Data return format — For the 'DataReturnFormat' database preference, these values are not recommended:
 - 'numeric'
 - 'cellarray'
 - 'structure'
- Error handling — The 'ErrorHandling' database preference is not recommended.
- Missing data — The 'NullNumberWrite', 'NullStringWrite', and 'NullNumberRead' database preferences for handling NULL data values are not recommended.

There are no plans to remove the `setdbprefs` function at this time.

Update Code

To set the data return format in prior releases, you specified returning imported data as a numeric matrix by setting the 'DataReturnFormat' database preference to the value 'numeric'. For example:

```
setdbprefs('DataReturnFormat','numeric')
results = fetch(conn,sqlquery);
```

Now you can set the same value by using the 'DataReturnFormat' name-value pair argument of the `fetch` function.

```
results = fetch(conn,sqlquery,'DataReturnFormat','numeric');
```

Or, you can customize import options.

```
opts = databaseImportOptions(conn,tablename);
varnames = "quantity";
opts = setoptions(opts,varnames,'Type','int64');
```

To specify error handling in prior releases, you set the 'ErrorHandling' database preference to the value 'report' or 'store' by using the `setdbprefs` function. For example:

```
setdbprefs('ErrorHandling','store')
```

Now you specify error handling by using the 'ErrorHandling' name-value pair argument of the database function or the 'ErrorHandling' name-value pair argument of the executeSQLScript function.

```
conn = database(datasource,username,password,'ErrorHandling','store');
```

To specify the handling of missing data in prior releases, you set the 'NullNumberWrite' database preference to a specific value, for example. This table shows database preference settings that are not recommended and the functionality you can use instead.

Discouraged Functionality	Recommended Replacement
setdbprefs('NullNumberWrite','NaN')	data input argument of sqlwrite
setdbprefs('NullStringWrite','null')	data input argument of sqlwrite
setdbprefs('NullNumberRead','0')	SQLImportOptions object

See Also

clear | fetch | getdatasources | database | close

splitsqlquery

Split SQL query using paging

Syntax

```
querybasket = splitsqlquery(conn,sqlquery)
querybasket = splitsqlquery(conn,sqlquery,'SplitSize',splitsize)
```

Description

`querybasket = splitsqlquery(conn,sqlquery)` splits an SQL query into a basket of multiple SQL queries. By default, each SQL query in the basket returns 100,000 rows in a batch. The resulting number of SQL queries in the basket depends on the size of the original SQL query results.

`querybasket = splitsqlquery(conn,sqlquery,'SplitSize',splitsize)` specifies a custom batch size for the number of rows returned by each SQL query in the basket.

Examples

Access Large Data from SQL Query Using Database Toolbox™

Determine the minimum arrival delay using a large set of flight data stored in a database. Access the database in a serial MATLAB® environment.

Using the `splitsqlquery` function, you can split the original SQL query into multiple SQL page queries. Then, you can access large data in chunks by using the `fetch` function.

To run this example, you must configure a JDBC data source. For more information, see the `configureJDBCDataSource` function.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
username = "";
password = "";
conn = database(datasource,username,password);
```

Define an SQL query to select all columns from the `airlinesmall` table, which contains 123,523 rows and 29 columns.

```
sqlquery = 'SELECT * FROM airlinesmall';
```

Split the original SQL query into multiple page queries and display them.

```
querybasket = splitsqlquery(conn,sqlquery)
```

```
querybasket =
```

```
2x1 string array
```

```
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 0 ROWS FETCH NEXT 100000  
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 100000 ROWS FETCH NEXT 200000
```

The query basket contains the page queries in a string array. The `splitsqlquery` function splits the queries using the default number of rows (100,000).

Define the `airlinesdata` variable.

```
airlinesdata = [];
```

Define the minimum arrival delay `minArrDelay` variable.

```
minArrDelay = [];
```

Execute the SQL page queries in `querybasket` by using a `for` loop, and import the data in chunks. Execute SQL page queries in the query basket, and import large data using the `fetch` function. Find and store the local minimum arrival delay for each chunk.

```
for i = 1: length(querybasket)  
  
    local_airlinesdata = fetch(conn,querybasket(i));  
  
    local_minArrDelay = min(local_airlinesdata.ArrDelay);  
  
    minArrDelay = [minArrDelay; local_minArrDelay];  
  
end
```

Find the minimum arrival delay from all the stored delays.

```
minArrDelay = min(minArrDelay)
```

```
minArrDelay =
```

```
-64
```

Close the database connection.

```
close(conn)
```

Access Large Data from SQL Query Using Database Toolbox and Parallel Computing Toolbox

Determine the minimum arrival delay using a large set of flight data stored in a database. Access the database using a parallel pool.

To initialize a parallel pool with a JDBC database connection, you must configure a JDBC data source. For more information, see the `configureJDBCDataSource` function.

Using the `splitsqlquery` function, you can split the original SQL query into multiple SQL page queries. Then, you can access large data in chunks by executing each SQL page query on a separate worker in the pool.

When you import large data, the performance depends on the SQL query, amount of data, machine specifications, and type of data analysis. To manage the performance, use the `splitsqlquery` function.

If you have a MATLAB® Parallel Server™ license, then use the `parpool` (Parallel Computing Toolbox) function with the cluster profile of your choice instead of the `gcp` (Parallel Computing Toolbox) function.

Create a database connection to the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
username = "";
password = "";
conn = database(datasource,username,password);
```

Define an SQL query to select all columns from the `airlinesmall` table, which contains 123,523 rows and 29 columns.

```
sqlquery = 'SELECT * FROM airlinesmall';
```

Split the original SQL query into multiple page queries and display them. Specify a split size of 10,000 rows.

```
splitsize = 10000;
querybasket = splitsqlquery(conn,sqlquery,'SplitSize',splitsize)
```

```
querybasket =
```

```
13x1 string array
```

```
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 0 ROWS FETCH NEXT 10000 R
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 10000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 20000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 30000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 40000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 50000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 60000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 70000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 80000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 90000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 100000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 110000 ROWS FETCH NEXT 10
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 120000 ROWS FETCH NEXT 35
```

The query basket contains the page queries in a string array. Each SQL query in the basket, except the last one, returns 10,000 rows.

Close the database connection.

```
close(conn)
```

Start the parallel pool.

```
pool = gcp;
```

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

Initialize the parallel pool using the JDBC data source.

```
c = createConnectionForPool(pool,datasource,username,password);
```

Define the `airlinesdata` variable.

```
airlinesdata = [];
```

Define the minimum arrival delay `minArrDelay` variable.

```
minArrDelay = [];
```

Use the `parfor` function to parallelize data access using the query basket.

For each worker:

- Retrieve the database connection object.
- Execute the SQL page query from the query basket and import data locally.
- Find the local minimum arrival delay.
- Store the local minimum arrival delay.

```
parfor i = 1: length(querybasket)
    conn = c.Value;
    local_airlinesdata = fetch(conn,querybasket(i));
    local_minArrDelay = min(local_airlinesdata.ArrDelay);
    minArrDelay = [minArrDelay; local_minArrDelay];
end
```

Find the minimum arrival delay using the stored delays from each worker.

```
minArrDelay = min(minArrDelay)
```

```
minArrDelay =
```

```
-64
```

Close the parallel pool.

```
delete(pool)
```

Input Arguments

conn — Database connection
connection object

Database connection, specified as a `connection` object created with the `database` function, `connection` object created with the `mysql` function, `connection` object created with the `postgresql` function, or `sqlite` object.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar.

For information about the SQL query language, see the SQL Tutorial.

Example: `SELECT * FROM invoice` selects all columns and rows from the `invoice` table.

Data Types: `char` | `string`

splitsize — SQL query split size

100000 (default) | numeric scalar

SQL query split size, specified as a numeric scalar. Specify this number to split an SQL query into a custom number of rows for each batch.

If the total number of rows returned from the original SQL query is less than 100,000 (the default), then the `splitsqlquery` function returns the original SQL query. Use this input argument to specify a smaller number of rows in a batch.

Data Types: `double`

Output Arguments

querybasket — SQL query basket

string array

SQL query basket, returned as a string array. Each SQL query in the basket is returned as a string scalar in the string array.

You can execute each SQL query in the basket using the `fetch` function. Or, you can run a parallel pool and assign each SQL query to a worker for execution.

Limitations

- The `splitsqlquery` function supports these databases only:
 - Microsoft SQL Server 2012 and later
 - Oracle
 - MySQL
 - PostgreSQL
 - SQLite
 - Amazon Redshift®
 - Amazon Aurora®
 - Google® Cloud SQL that runs an instance of MySQL or PostgreSQL
 - MariaDB

If the `connection` object uses an unsupported database, the `splitsqlquery` function displays a warning and returns the original SQL query.

- The `splitsqlquery` function does not support the MATLAB interface to SQLite.

Version History

Introduced in R2017b

See Also

`fetch` | `database` | `close` | `javaaddpath` | `addAttachedFiles` | `parpool` | `parfevalOnAll` | `parallel.pool.Constant`

Topics

“Analyze Large Data in Database Using Tall Arrays” on page 5-30

External Websites

SQL Tutorial

createConnectionForPool

Initialize parallel pool using database connection

Syntax

```
c = createConnectionForPool(pool,datasource,username,password)
```

Description

`c = createConnectionForPool(pool,datasource,username,password)` initializes a parallel pool by creating a database connection on each worker in the pool using a data source, user name, and password.

Note If you use an ODBC data source for the database connection, each worker in the pool must have the required ODBC driver installed and a configured ODBC data source. Otherwise, an error occurs when you import data from the database.

Examples

Initialize Parallel Pool with ODBC Database Connection

Using an ODBC database connection, access a database using a parallel pool (requires Parallel Computing Toolbox™). Import data from multiple SQL queries in parallel.

Each worker in the pool has the ODBC driver installed. Also, each worker has a configured ODBC data source. For more information, see the `configureODBCDataSource` function.

Start the parallel pool.

```
pool = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Initialize the parallel pool using an ODBC data source. This data source configures an ODBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MS SQL Server Auth";
username = "";
password = "";
c = createConnectionForPool(pool,datasource,username,password);
```

Define the SQL queries.

```
sqlqueries = ["SELECT * FROM invoice" ...
              "SELECT * FROM inventorytable" ...
              "SELECT * FROM producttable"];
```

Parallelize data access using the query basket by calling the `parfor` function.

For each worker, retrieve the database connection object, execute the SQL queries, and import data locally.

```
parfor i = 1:length(sqlqueries)
    conn = c.Value;
    results = fetch(conn,sqlqueries(i));
    allresults{i} = results;
```

```
end
```

Display the results for all queries. The cell array contains three tables, one for each set of query results.

```
allresults
allresults = 1x3 cell array
    {10x5 table}    {13x4 table}    {15x5 table}
```

Close the parallel pool.

```
delete(pool);
```

Initialize Parallel Pool with JDBC Database Connection

Using a JDBC database connection, access a database using a parallel pool (requires Parallel Computing Toolbox™). Import data from multiple SQL queries in parallel.

To initialize a parallel pool with a JDBC database connection, you must configure a JDBC data source. For more information, see the `databaseConnectionOptions` function.

Start the parallel pool.

```
pool = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Initialize the parallel pool using the JDBC data source `MSSQLServerJDBCAuth`. This data source configures a JDBC driver to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MSSQLServerJDBCAuth";
username = "";
password = "";
c = createConnectionForPool(pool,datasource,username,password);
```

Define the SQL queries.

```
sqlqueries = ["SELECT * FROM invoice" ...
    "SELECT * FROM inventorytable" ...
    "SELECT * FROM producttable"];
```

Parallelize data access using the query basket by calling the `parfor` function.

For each worker, retrieve the database connection object, execute the SQL queries, and import data locally.

```
parfor i = 1:length(sqlqueries)
    conn = c.Value;
    results = fetch(conn,sqlqueries(i));
    allresults{i} = results;
end
```

Display the results for all queries. The cell array contains three tables, one for each set of query results.

```
allresults
allresults = 1x3 cell array
    {10x5 table}    {13x4 table}    {15x5 table}
```

Close the parallel pool.

```
delete(pool);
```

Input Arguments

pool — Parallel pool

`parallel.Pool` object

Parallel pool, specified as a `parallel.Pool` object.

datasource — Data source name

character vector | string scalar

Data source name, specified as a character vector or string scalar. Specify the name of an existing data source.

Example: "myDataSource"

Data Types: `char` | `string`

username — User name

character vector | string scalar

User name required to access the database, specified as a character vector or string scalar. If no user name is required, specify an empty value "".

Data Types: `char` | `string`

password — Password

character vector | string scalar

Password required to access the database, specified as a character vector or string scalar. If no password is required, specify an empty value "".

Data Types: `char` | `string`

Output Arguments

c — `parallel.pool.Constant`

`parallel.pool.Constant` object

`parallel.pool.Constant`, specified as a `parallel.pool.Constant` object. The `Value` property of the `parallel.pool.Constant` object is available only on workers.

Version History

Introduced in R2019a

See Also

`database` | `splitsqlquery`

Topics

“Configure Driver and Data Source” on page 2-14

sqlite

SQLite connection

Description

The `sqlite` function creates an `sqlite` object. You can use this object to connect to an SQLite database file using the MATLAB interface to SQLite. The MATLAB interface to SQLite enables you to work with SQLite database files without installing and administering a database or driver. For details, see “Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5.

Creation

Syntax

```
conn = sqlite(dbfile)
conn = sqlite(dbfile,mode)
```

Description

`conn = sqlite(dbfile)` connects to an existing SQLite database file.

`conn = sqlite(dbfile,mode)` connects to an existing database file or creates and connects to a new database file, depending on the mode type.

Input Arguments

dbfile — SQLite database file

character vector | string scalar

SQLite database file, specified as a character vector or string scalar. You can use the database file to store data and import and export it to MATLAB.

Data Types: `char` | `string`

mode — SQLite database file mode

"connect" (default) | "readonly" | "create"

SQLite database file mode, specified as one of these values.

Value	Description
"connect"	Connect to an existing SQLite database file.
"readonly"	Create a read-only connection to an existing SQLite database file.
"create"	Create and connect to a new SQLite database file.

The file mode determines whether you connect to an existing SQLite database file or create a new one. For existing database files, the file mode determines whether the database connection is read-only and sets the `IsReadOnly` property. You can specify the file mode as a string scalar or character vector.

Properties

Database — SQLite database file name

character vector

This property is read-only.

SQLite database file name, specified as a character vector that contains the full path to the SQLite database file.

The `dbfile` input argument sets this property.

Example: `'C:\tutorial.db'`

Data Types: `char`

IsOpen — Database connection indicator

0 (default) | 1

This property is read-only.

Database connection indicator, specified as a logical 0 when the database connection is closed or invalid, or a logical 1 when the database connection is open. This property is hidden from the display.

IsReadOnly — Read-only database file indicator

0 (default) | 1

This property is read-only.

Read-only database file indicator, specified as a logical 0 when the SQLite database file can be modified, or a logical 1 when the database file is read-only.

Data Types: `logical`

AutoCommit — Flag to autocommit transactions

'on' (default) | 'off'

Flag to autocommit transactions, specified as one of these values:

- 'on' — Database transactions are automatically committed to the database.
- 'off' — Database transactions must be committed to the database manually.

Object Functions

SQLite Database Connection

`isopen` Determine if SQLite connection is open

`close` Close SQLite connection

Import Data into MATLAB

`sqlread` Import data into MATLAB from SQLite database table
`fetch` Import data into MATLAB workspace using SQLite connection

Export Data from MATLAB

`sqlwrite` Insert MATLAB data into SQLite database table

Database Operations

`commit` Make changes to SQLite database file permanent
`execute` Execute SQL statement using SQLite database connection
`rollback` Undo changes to SQLite database file
`sqlupdate` Update rows in SQLite database table

Examples

Create SQLite Connection to Existing Database File

Create an SQLite connection to the MATLAB® interface to SQLite using the existing database file `tutorial.db`. Specify the file name in the current folder.

```
dbfile = fullfile(pwd,"tutorial.db");
conn = sqlite(dbfile)
```

```
conn =
  sqlite with properties:
```

```
    Database: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp64b75b10\database-ex96650978\tutorial.db'
    IsReadOnly: 0
    AutoCommit: 'on'
```

`conn` is an `sqlite` object with these properties:

- `Database` — SQLite database file name.
- `IsOpen` — SQLite connection is open.
- `IsReadOnly` — SQLite connection is writable.

To import data from the database file, you can use the `fetch` function.

Close the SQLite connection.

```
close(conn)
```

Create SQLite Connection Using New Database File

Create an SQLite connection to the MATLAB® interface to SQLite using a new database file named `mysqlite.db`. Specify the file name in the current folder.

```
dbfile = fullfile(pwd,"mysqlite.db");
conn = sqlite(dbfile,"create")
```

```
conn =
    sqlite with properties:

        Database: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp64b75b10\database-ex61952421\mysqlite'
        IsReadOnly: 0
        AutoCommit: 'on'
```

`conn` is an `sqlite` object with these properties:

- `Database` — SQLite database file name.
- `IsOpen` — SQLite connection is open.
- `IsReadOnly` — SQLite connection is writable.

To insert data into the database file, use the `sqlwrite` function.

Close the SQLite connection.

```
close(conn)
```

Create Read-Only SQLite Connection

Create a read-only SQLite connection to the MATLAB® interface to SQLite using the existing database file `tutorial.db`. Specify the file name in the current folder.

```
dbfile = fullfile(pwd,"tutorial.db");
conn = sqlite(dbfile,"readonly")
```

```
conn =
    sqlite with properties:

        Database: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp64b75b10\database-ex41829813\tutorial'
        IsReadOnly: 1
        AutoCommit: 'on'
```

`conn` is an `sqlite` object with these properties:

- `Database` — SQLite database file name.
- `IsOpen` — SQLite connection is open.
- `IsReadOnly` — SQLite connection is read-only.

To import data from the database file, you can use the `fetch` function.

Close the SQLite connection.

```
close(conn)
```

Alternative Functionality

Instead of the `sqlite` object, the `connection` object enables you to connect to various relational databases using ODBC and JDBC drivers that you install and administer. You can create the

connection object by using the `database` function. To use the JDBC driver, close the SQLite connection and create a database connection using the URL string. For details, see these topics depending on your platform:

- “SQLite JDBC for Windows” on page 2-65
- “SQLite JDBC for macOS” on page 2-110
- “SQLite JDBC for Linux” on page 2-114

Version History

Introduced in R2016a

See Also

Topics

“Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5

“Import Data Using MATLAB Interface to SQLite” on page 5-32

“Insert Data into SQLite Database Table” on page 5-36

“Deploy MATLAB Interface to SQLite Database Application with MATLAB Compiler” on page 5-43

close

Close SQLite connection

Syntax

```
close(conn)
```

Description

`close(conn)` closes the SQLite connection by using the MATLAB interface to SQLite.

Note The SQLite connection object remains open until you close it using the `close` function. Always close this object when you finish using it.

Examples

Close SQLite Connection Object

Create an SQLite connection using the MATLAB® interface to SQLite and the existing database file `tutorial.db`, which is in the current folder.

```
dbfile = fullfile(pwd,"tutorial.db");  
conn = sqlite(dbfile);
```

To import data from the database file, use the `fetch` function.

Close the SQLite connection.

```
close(conn)
```

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

Version History

Introduced in R2016a

See Also

Objects

sqlite

Functions

sqlread | fetch | sqlwrite

Topics

“Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5

“Import Data Using MATLAB Interface to SQLite” on page 5-32

“Insert Data into SQLite Database Table” on page 5-36

“Create Table and Add Column in SQLite Database” on page 5-38

fetch

Import data into MATLAB workspace using SQLite connection

Syntax

```
results = fetch(conn,sqlquery)
results = fetch(conn,sqlquery,Name=Value)
```

Description

`results = fetch(conn,sqlquery)` returns all rows of data from an SQLite database file immediately after executing the SQL statement `sqlquery` by using the SQLite connection `conn` of the MATLAB interface to SQLite.

`results = fetch(conn,sqlquery,Name=Value)` specifies additional options using one or more name-value arguments. For example, `MaxRows=5` imports five rows of data.

Examples

Import Data from Database Table in SQLite Database File

Import all rows of data from a database table in an SQLite database file into MATLAB®. Determine the highest unit cost among products in the table. Then, use a row filter to import only the data for products with a unit cost less than 15.

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = 'tutorial.db';
conn = sqlite(dbfile);
```

Import all the data from `productTable`. The `results` output argument contains the imported data as a table.

```
sqlquery = 'SELECT * FROM productTable';
results = fetch(conn,sqlquery)
```

`results=15x5 table`

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	125970	1003	13	"Victorian Doll"
8	212569	1001	5	"Train Set"
7	389123	1007	16	"Engine Kit"
2	400314	1002	9	"Painting Set"
4	400339	1008	21	"Space Cruiser"
1	400345	1001	14	"Building Blocks"
5	400455	1005	3	"Tin Soldier"
6	400876	1004	8	"Sail Boat"
3	400999	1009	17	"Slinky"

10	888652	1006	24	"Teddy Bear"
11	408143	1004	11	"Convertible"
12	210456	1010	22	"Hugsy"
13	470816	1012	16	"Pancakes"
14	510099	1011	19	"Shawl"
15	899752	1011	20	"Snacks"

Determine the highest unit cost of the products. Access the variable in the table for the unit cost data, and then find the maximum cost.

```
max(results.unitCost)
```

```
ans = int64
      24
```

Now, import the data using a row filter. The filter condition is that `unitCost` must be less than 15.

```
rf = rowfilter("unitCost");
rf = rf.unitCost < 15;
results = fetch(conn,sqlquery,"RowFilter",rf)
```

```
results=7x5 table
      productNumber  stockNumber  supplierNumber  unitCost  productDescription
      _____  _____  _____  _____  _____
           9         125970         1003          13      "Victorian Doll"
           8         212569         1001           5      "Train Set"
           2         400314         1002           9      "Painting Set"
           1         400345         1001          14      "Building Blocks"
           5         400455         1005           3      "Tin Soldier"
           6         400876         1004           8      "Sail Boat"
          11         408143         1004          11      "Convertible"
```

Close the SQLite connection.

```
close(conn)
```

Limit Number of Rows in Imported Data

Use the MATLAB® interface to SQLite to import a limited number of rows into MATLAB from a database table in an SQLite database file. Then, determine the highest unit cost among products in the table.

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Import five rows of data from `productTable` by using the `MaxRows` name-value argument. `results` contains five rows of imported data as a table.

```
sqlquery = "SELECT * FROM productTable";
results = fetch(conn,sqlquery,MaxRows=5)
```

```

results=5x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
  _____  _____  _____  _____  _____
          9      125970      1003          13      "Victorian Doll"
          8      212569      1001           5      "Train Set"
          7      389123      1007          16      "Engine Kit"
          2      400314      1002           9      "Painting Set"
          4      400339      1008          21      "Space Cruiser"

```

Determine the highest unit cost for the limited number of products. Access the variable in the table for the unit cost data, and then find the maximum cost.

```

data = results.unitCost;
max(data)

```

```

ans = int64
     21

```

Close the SQLite connection.

```
close(conn)
```

Copyright 2021 The MathWorks, Inc.

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar. The SQL statement can be any valid SQL statement, including nested queries. For information about the SQL query language, see the SQL Tutorial.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `fetch(conn,sqlquery,MaxRows=5)` imports five rows of data.

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as a positive numeric scalar. By default, the `fetch` function returns all rows from the executed SQL query. Use this name-value argument to limit the number of rows imported into MATLAB.

Example: MaxRows=10

Data Types: double

VariableNamingRule — Variable naming rule

"preserve" (default) | "modify"

Variable naming rule, specified as one of these values:

- "preserve" — Preserve most variable names when the `fetch` function imports data.
- "modify" — Remove non-ASCII characters from variable names when the `fetch` function imports data.

Example: VariableNamingRule="modify"

Data Types: string

RowFilter — Row filter condition

<unconstrained> (default) | matlab.io.RowFilter object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; fetch(conn,sqlquery,"RowFilter",rf)`

Output Arguments

results — Result data

table

Result data, returned as a table. The result data contains all rows of data from the executed SQL statement.

The `fetch` function converts SQLite data types to MATLAB data types and represents NULL values accordingly.

SQLite Data Type	MATLAB Data Type	MATLAB Null Value Representation
<ul style="list-style-type: none"> • REAL • DOUBLE • FLOAT • NUMERIC • INT • TINYINT • SMALLINT • MEDIUMINT • BIGINT 	double	double(NaN)
<ul style="list-style-type: none"> • CHAR • VARCHAR 	string	<missing>

SQLite Data Type	MATLAB Data Type	MATLAB Null Value Representation
<ul style="list-style-type: none"> DATE DATETIME 	string	<missing>
<ul style="list-style-type: none"> BLOB 	$N \times 1$ uint8 vector	0×1 uint8 vector
<ul style="list-style-type: none"> BOOLEAN 	int64	Not available

Version History

Introduced in R2016a

R2023a: Selectively import rows of data based on filter condition

You can use the `RowFilter` name-value argument to selectively import rows of data from a database table.

R2022a: fetch function returns table

Behavior changed in R2022a

In prior releases, the `fetch` function returned the `results` output argument as a cell array. In R2022a, the `fetch` function returns the `results` output argument as a table. Use the `table2cell` function to convert the data type back to a cell array, or adjust your code to accept the new data type.

See Also

Objects

`sqlite`

Functions

`close` | `execute` | `sqlwrite`

Topics

“Interact with Data in SQLite Database Using MATLAB Interface to SQLite” on page 2-5

“Import Data Using MATLAB Interface to SQLite” on page 5-32

“Insert Data into SQLite Database Table” on page 5-36

External Websites

SQL Tutorial

SQLite Home Page

update

Package: database.odbcc

Replace data in database table with MATLAB data

Syntax

```
update(conn,tablename,colnames,data,whereclause)
```

Description

`update(conn,tablename,colnames,data,whereclause)` exports the MATLAB variable `data` in its current format into the database table `tablename` using the database connection `conn`. You can use the SQL `WHERE` statement to specify which existing records in the database to replace.

Examples

Update Existing Record Using Cell Array

Connect to a Microsoft Access database and store the data that you are updating in a cell array. Then, update one column of data in the database table. Close the database connection.

Create the database connection `conn` to the Microsoft Access database. This code assumes that you are connecting to a data source named `dbdemo` with a blank user name and password.

```
conn = database('dbdemo','','');
```

This database contains the table `inventorytable`, which contains these columns:

- `productnumber`
- `quantity`
- `price`
- `inventorydate`

Import all the data from `inventorytable` as a cell array by using `conn`, and display the first three rows of imported data.

```
sqlquery = 'SELECT * FROM inventorytable';
results = fetch(conn,sqlquery,'DataReturnFormat','cellarray');
results(1:3,:)
```

```
ans =
```

```
3×4 cell array
```

```

    {[1]}    {[1700]}    {[15]}    {'2014-09-23 09:3...'}
    {[2]}    {[1200]}    {[ 9]}    {'2014-07-08 22:5...'}
    {[3]}    {[ 356]}    {[17]}    {'2014-05-14 07:1...'}

```

Define a cell array containing the name of the column that you are updating.

```
colnames = {'quantity'};
```

Define a cell array containing the new data, 2000.

```
data = {2000};
```

Update the column `quantity` in `inventorytable` for the product with `productnumber` equal to 1.

```
tablename = 'inventorytable';
whereclause = 'WHERE productnumber = 1';
```

```
update(conn,tablename,colnames,data,whereclause)
```

Import the data again and view the updated contents in `inventorytable`.

```
results = fetch(conn,sqlquery,'DataReturnFormat','cellarray');
results(1:3,:)
```

```
ans =
```

```
3×4 cell array
```

```

    {[1]}    {[2000]}    {[15]}    {'2014-09-23 09:3...'}
    {[2]}    {[1200]}    {[ 9]}    {'2014-07-08 22:5...'}
    {[3]}    {[ 356]}    {[17]}    {'2014-05-14 07:1...'}

```

In the `inventorytable` data, the product with the product number equal to 1 has an updated quantity of 2000 units.

Close the database connection.

```
close(conn)
```

Update Existing Record Using Table

Connect to a Microsoft Access database and store the data that you are updating as a table. Then, update multiple columns of data in the database table. Close the database connection.

Create the database connection `conn` to the Microsoft Access database. This code assumes that you are connecting to a data source named `dbdemo` with a blank user name and password.

```
conn = database('dbdemo','','');
```

This database contains the table `inventorytable`, which contains these columns:

- `productnumber`
- `quantity`
- `price`
- `inventorydate`

Import all the data from `inventorytable` by using `conn`, and display a few rows of the imported data.

```
sqlquery = 'SELECT * FROM inventorytable';
results = fetch(conn,sqlquery);
head(results)
```

```
ans =
```

```
8x4 table
```

productnumber	quantity	price	inventorydate
1	1700	20	'2014-09-23 09:38:34.000'
2	1200	9	'2014-07-08 22:50:45.000'
3	356	17	'2014-05-14 07:14:28.000'
4	2580	21	'2013-06-08 14:24:33.000'
5	9000	3	'2012-09-14 15:00:25.000'
6	4540	8	'2013-12-25 19:45:00.000'
7	6034	16	'2014-08-06 08:38:00.000'
8	8350	5	'2011-06-18 11:45:35.000'

Define a cell array containing the names of the columns that you are updating in `inventorytable`.

```
colnames = {'price','inventorydate'};
```

Define a table that contains the new data. Update the price to \$15 and set the inventory timestamp to '2014-12-01 08:50:15.000'.

```
data = table(15,{'2014-12-01 08:50:15.000'}, ...
    'VariableNames',{'price','inventorydate'});
```

Update the columns `price` and `inventorydate` in the table `inventorytable` for the product number equal to 1.

```
tablename = 'inventorytable';
whereclause = 'WHERE productnumber = 1';
```

```
update(conn,tablename,colnames,data,whereclause)
```

Import the data again and view the updated contents in `inventorytable`.

```
results = fetch(conn,sqlquery);
head(results)
```

```
ans =
```

```
8x4 table
```

productnumber	quantity	price	inventorydate
1	1700	15	'2014-12-01 08:50:15.000'
2	1200	9	'2014-07-08 22:50:45.000'
3	356	17	'2014-05-14 07:14:28.000'
4	2580	21	'2013-06-08 14:24:33.000'
5	9000	3	'2012-09-14 15:00:25.000'
6	4540	8	'2013-12-25 19:45:00.000'
7	6034	16	'2014-08-06 08:38:00.000'
8	8350	5	'2011-06-18 11:45:35.000'

The product with the product number equal to 1 has an updated price of \$15 and timestamp of '2014-12-01 08:50:15.000'.

Close the database connection.

```
close(conn)
```

Update Multiple Records with Multiple Conditions

Connect to a Microsoft Access database and store the data that you are updating in a cell array. Then, update multiple records of data in the table by using multiple `WHERE` clauses. Close the database connection.

Create the database connection `conn` to the Microsoft Access database. This code assumes that you are connecting to a data source named `dbdemo` with a blank user name and password.

```
conn = database('dbdemo', '', '');
```

This database contains the table `inventorytable`, which contains these columns:

- `productnumber`
- `quantity`
- `price`
- `inventorydate`

Import all the data from `inventorytable` by using `conn`, and display the first few rows of imported data.

```
sqlquery = 'SELECT * FROM inventorytable';
results = fetch(conn,sqlquery);
head(results)
```

```
ans =
```

```
8×4 table
```

<u>productnumber</u>	<u>quantity</u>	<u>price</u>	<u>inventorydate</u>
1	1700	20	'2014-12-01 08:50:15.000'
2	1200	9	'2014-07-08 22:50:45.000'
3	356	17	'2014-05-14 07:14:28.000'
4	2580	21	'2013-06-08 14:24:33.000'
5	9000	3	'2012-09-14 15:00:25.000'
6	4540	8	'2013-12-25 19:45:00.000'
7	6034	16	'2014-08-06 08:38:00.000'
8	8350	5	'2011-06-18 11:45:35.000'

Define a cell array containing the name of the column that you are updating.

```
colnames = {'quantity'};
```

Define a cell array containing the new data. Update the quantities for two products.

```
A = 10000; % new quantity for product number 5
B = 5000; % new quantity for product number 8
```

```
data = {A;B}; % cell array with the new quantities
```

Update the column `quantity` in `inventorytable` for the products with product numbers equal to 5 and 8. Create a cell array `whereclause` that contains two `WHERE` clauses, one for each product.

```
tablename = 'inventorytable';
whereclause = {'WHERE productnumber = 5'; 'WHERE productnumber = 8'};

update(conn, tablename, colnames, data, whereclause)
```

Import the data again and view the updated contents in `inventorytable`.

```
results = fetch(conn, sqlquery);
head(results)
```

ans =

8×4 table

productnumber	quantity	price	inventorydate
1	1700	20	'2014-12-01 08:50:15.000'
2	1200	9	'2014-07-08 22:50:45.000'
3	356	17	'2014-05-14 07:14:28.000'
4	2580	21	'2013-06-08 14:24:33.000'
5	10000	3	'2012-09-14 15:00:25.000'
6	4540	8	'2013-12-25 19:45:00.000'
7	6034	16	'2014-08-06 08:38:00.000'
8	5000	5	'2011-06-18 11:45:35.000'

The product with the product number equal to 5 has an updated quantity of 10000 units. The product with the product number equal to 8 has an updated quantity of 5000 units.

Close the database connection.

```
close(conn)
```

Update Multiple Columns with Multiple Conditions

Connect to a Microsoft Access database and store the data that you are updating in a cell array. Then, update multiple columns of data in the table by using multiple `WHERE` clauses. Close the database connection.

Create the database connection `conn` to the Microsoft Access database. This code assumes that you are connecting to a data source named `dbdemo` with a blank user name and password.

```
conn = database('dbdemo', '', '');
```

This database contains the table `inventorytable`, which contains these columns:

- `productnumber`
- `quantity`
- `price`
- `inventorydate`

Import all the data from `inventorytable` by using `conn`, and display the first few rows of imported data.

```
sqlquery = 'SELECT * FROM inventorytable';
results = fetch(conn,sqlquery);
head(results)
```

```
ans =
```

```
8×4 table
```

productnumber	quantity	price	inventorydate
1	1700	20	'2014-12-01 08:50:15.000'
2	1200	9	'2014-07-08 22:50:45.000'
3	356	17	'2014-05-14 07:14:28.000'
4	2580	21	'2013-06-08 14:24:33.000'
5	9000	3	'2012-09-14 15:00:25.000'
6	4540	8	'2013-12-25 19:45:00.000'
7	6034	16	'2014-08-06 08:38:00.000'
8	8350	5	'2011-06-18 11:45:35.000'

Define a cell array containing the names of the columns that you are updating.

```
colnames = {'quantity','price'};
```

Define a cell array containing the new data. Update the quantities and prices for two products.

```
% new quantities and prices for product numbers 5 and 8
% are separated by a semicolon in the cell array
data = {10000,5.5;9000,10};
```

Update the columns quantity and price in inventorytable for the products with product numbers equal to 5 and 8. Create a cell array whereclause that contains two WHERE clauses, one for each product.

```
tablename = 'inventorytable';
whereclause = {'WHERE productnumber = 5';'WHERE productnumber = 8'};
```

```
update(conn,tablename,colnames,data,whereclause)
```

Import the data again and view the updated contents in inventorytable.

```
results = fetch(conn,sqlquery);
head(results)
```

```
ans =
```

```
8×4 table
```

productnumber	quantity	price	inventorydate
1	1700	20	'2014-12-01 08:50:15.000'
2	1200	9	'2014-07-08 22:50:45.000'
3	356	17	'2014-05-14 07:14:28.000'
4	2580	21	'2013-06-08 14:24:33.000'
5	10000	6	'2012-09-14 15:00:25.000'
6	4540	8	'2013-12-25 19:45:00.000'
7	6034	16	'2014-08-06 08:38:00.000'
8	9000	10	'2011-06-18 11:45:35.000'

The product with the product number equal to 5 has an updated quantity of 10000 units and price equal to 6, rounded to the nearest number. The product with the product number equal to 8 has an updated quantity of 9000 units and price equal to 10.

Close the database connection.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

colnames — Database table column names

cell array of character vectors | string array

Database table column names, specified as a cell array of one or more character vectors or a string array to denote the columns in the existing database table `tablename`.

Example: {'col1', 'col2', 'col3'}

Data Types: cell | string

data — Update data

cell array | numeric matrix | table | structure | dataset

Update data, specified as a cell array, numeric matrix, table, structure, or dataset array.

If you are connecting to a database using a JDBC driver, convert the update data to a supported format before running `update`. If `data` contains MATLAB dates, times, or timestamps, use this formatting:

- Dates must be character vectors of the form `yyyy-mm-dd`.
- Times must be character vectors of the form `HH:MM:SS`.
- Timestamps must be character vectors of the form `yyyy-mm-dd HH:MM:SS.FFF`.

The database preference settings `NullNumberWrite` and `NullStringWrite` do not apply to this function. If `data` contains `null` entries and NaNs, convert these entries to an empty value `''`.

- If `data` is a structure, then field names in the structure must match `colnames`.
- If `data` is a table or a dataset array, then the variable names in the table or dataset array must match `colnames`.

whereclause — SQL WHERE clause

character vector | cell array of character vectors | string scalar | string array

SQL WHERE clause, specified as a character vector or string scalar for one condition or a cell array of character vectors or string array for multiple conditions.

Example: 'WHERE producttable.productnumber = 1'

Data Types: char | cell | string

Tips

- The value of the `AutoCommit` property in the connection object determines whether update automatically commits the data to the database.
 - To view the `AutoCommit` value, access it using the connection object; for example, `conn.AutoCommit`.
 - To set the `AutoCommit` value, use the corresponding name-value pair argument in the database function.
 - To commit the data to the database, use the `commit` function or issue an SQL COMMIT statement using the `exec` function.
 - To roll back the data, use `rollback` or issue an SQL ROLLBACK statement using the `exec` function.
- You can use `datainsert` to add new rows instead of replacing existing data.
- To update multiple records, the number of SQL WHERE clauses in `whereclause` must match the number of records in `data`.
- If the order of records in your database is not constant, then you can use values of column names to identify records.
- If this error message appears when your database table is open in edit mode:

```
[Vendor][ODBC Product Driver] The database engine could
not lock table 'TableName' because it is already in use
by another person or process.
```

Then, close the table and rerun the update function.

- Running the same update operation again can cause this error message to appear.

```
??? Error using ==> database.update
Error:Commit/Rollback Problems
```

Version History

Introduced before R2006a

See Also

`commit` | `database` | `datainsert` | `rollback` | `get` | `close`

Topics

“Replace Existing Data in Database” on page 5-12

“Roll Back Data After Updating Record” on page 5-9

“Import Data from Database Table Using `sqlread` Function” on page 5-58

“Data Type Support” on page 1-3

External Websites

SQL Tutorial

sqlfind

Package: database.odbc

Find information about all table types in database

Syntax

```
data = sqlfind(conn,pattern)
data = sqlfind(conn,pattern,Name,Value)
```

Description

`data = sqlfind(conn,pattern)` returns information about all the “Table Types” on page 11-349 in a database where the specified character pattern appears in the name of a table type. Executing this function is the equivalent of writing the SQL statement `SELECT * FROM information_schema.tables`.

`data = sqlfind(conn,pattern,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, 'Catalog', 'cat' finds all table types in the 'cat' catalog.

Examples

Find Information About Table Types in Database

Use an ODBC connection to find information about all database table types in a Microsoft® SQL Server® database.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
    []
```

Find information about all table types in the database.

```
data = sqlfind(conn, '');
```

Display information about the first three table types.

```
data(1:3, :)
```

```
ans =
```

```
3x5 table
```

Catalog	Schema	Table	Columns	Type
'toy_store'	'INFORMATION_SCHEMA'	'CHECK_CONSTRAINTS'	{1x4 cell}	'VIEW'
'toy_store'	'INFORMATION_SCHEMA'	'COLUMNS'	{1x23 cell}	'VIEW'
'toy_store'	'INFORMATION_SCHEMA'	'COLUMN_DOMAIN_USAGE'	{1x7 cell}	'VIEW'

data contains these variables:

- Catalog name
- Schema name
- Table name
- Columns in the table type
- Table type

Close the database connection.

```
close(conn)
```

Find Information About Table

Use an ODBC connection to find information about a database table in a Microsoft® SQL Server® database.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Find information about any tables that contain the pattern `product` in the table name. The `sqlfind` function returns information about the table `productTable`.

```
pattern = 'product';
data = sqlfind(conn,pattern)
```

```
data =
```

```
1x5 table
```

Catalog	Schema	Table	Columns	Type
'toy_store'	'dbo'	'productTable'	{1x5 cell}	'TABLE'

data contains these variables:

- Catalog name
- Schema name
- Table name
- Columns in the database table
- Table type

Display the column names in productTable.

```
data.Columns{:}
```

```
ans =
```

```
1x5 cell array
```

```
Columns 1 through 4
```

```
{'productNumber'} {'stockNumber'} {'supplierNumber'} {'unitCost'}
```

```
Column 5
```

```
{'productDescript...'}

```

Close the database connection.

```
close(conn)
```

Find Information About Table Types in Catalog and Schema

Use an ODBC connection to find information about all database table types in a Microsoft® SQL Server® database. Specify the database catalog and schema to search.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the Message property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Find information about all table types in the `toy_store` database catalog and the `dbo` database schema. Use the `'Catalog'` name-value pair argument to specify the catalog. Use the `'Schema'` name-value pair argument to specify the schema.

`data` is a table that contains information about all the table types in the specified catalog and schema.

```
data = sqlfind(conn, '', 'Catalog', 'toy_store', 'Schema', 'dbo');
```

Display the first eight table types.

```
head(data)
```

```
ans =
```

```
8x5 table
```

Catalog	Schema	Table	Columns	Type
'toy_store'	'dbo'	'DS17111713025590'	{1x5 cell}	'TABLE'
'toy_store'	'dbo'	'DS17111713025699'	{1x4 cell}	'TABLE'
'toy_store'	'dbo'	'DS22121715025751'	{1x5 cell}	'TABLE'
'toy_store'	'dbo'	'DS22121715025879'	{1x4 cell}	'TABLE'
'toy_store'	'dbo'	'DS22121715052820'	{1x5 cell}	'TABLE'
'toy_store'	'dbo'	'DS22121715052941'	{1x4 cell}	'TABLE'
'toy_store'	'dbo'	'DS26121710493780'	{1x5 cell}	'TABLE'
'toy_store'	'dbo'	'DS26121710493818'	{1x4 cell}	'TABLE'

`data` contains these variables:

- Catalog name
- Schema name
- Table name
- Columns in the database table
- Table type

Display the column names in the fourth table type.

```
data.Columns{4}
```

```
ans =
```

```
1x4 cell array
```

```
 {'productNumber'} {'Quantity'} {'Price'} {'inventoryDate'}
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

pattern — Pattern

character vector | string scalar

Pattern, specified as a character vector or string scalar. The `sqlfind` function searches for this text in the names of the tables types in a database.

Example: "inventory"

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `data = sqlfind(conn,pattern,'Catalog','toy_store','Schema','dbo')` returns information about table types, stored in the specified catalog and schema, that match the name of the table type with the specified pattern.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: string | char

Schema — Database schema name

string scalar | character vector

Database schema name, specified as a string scalar or character vector. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Example: `Schema = "dbo"`

Data Types: string | char

Output Arguments

data — Table type information

table

Table type information, returned as a table that contains information for table types, where the table type name partially or fully matches the text in `pattern`. The returned table has these variables.

Variable	Description	Variable Data Type
Catalog	Catalog name where the database table type is stored	Cell array of character vectors
Schema	Schema name where the database table type is stored	
Table	Database table name	
Columns	Column names in the database table type	
Type	Database table type	

More About

Table Types

Table types are a subset of database objects, which store or reference data.

The `sqlfind` function recognizes these table types in a database:

- Table
- View
- System table
- System view
- Synonym
- Global temporary table
- Local temporary table

Version History

Introduced in R2018a

See Also

sqlread | sqlinnerjoin | database | close

Topics

“Retrieve Database Metadata” on page 5-64

“Import Data from Database Table Using sqlread Function” on page 5-58

sqlinnerjoin

Package: database.odbc

Inner join between two database tables

Syntax

```
data = sqlinnerjoin(conn,lefttable,righttable)
data = sqlinnerjoin(conn,lefttable,righttable,Name,Value)
```

Description

`data = sqlinnerjoin(conn,lefttable,righttable)` returns a table resulting from an inner join between the left and right database tables. This function matches rows using all shared columns, or keys, in both database tables. The inner join retains only the rows that match between the two tables. Executing this function is the equivalent of writing the SQL statement `SELECT * FROM lefttable,righttable INNER JOIN lefttable.key = righttable.key`.

`data = sqlinnerjoin(conn,lefttable,righttable,Name,Value)` uses additional options specified by one or more name-value arguments. For example, specify `Keys = "productNumber"` to use the `productNumber` column as a key for joining the two database tables.

Examples

Join Two Database Tables

Use an ODBC connection to import product data from an inner join between two Microsoft® SQL Server® database tables into MATLAB®.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlinnerjoin` function automatically detects the shared column between the tables.

`data` is a table that contains the matched rows from the two tables.

```
lefttable = 'productTable';
righttable = 'suppliers';
data = sqlinnerjoin(conn,lefttable,righttable);
```

Display the first three rows of matched data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,3)
```

```
ans =
```

```
3×10 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0035e+05	1001	14	'Building Blocks'	1001
2	4.0031e+05	1002	9	'Painting Set'	1002
3	4.01e+05	1009	17	'Slinky'	1009

Close the database connection.

```
close(conn)
```

Join Two Database Tables in Catalog and Schema

Use an ODBC connection to import product data from an inner join between two Microsoft® SQL Server® database tables into MATLAB®. Specify the database catalog and schema where the tables are stored.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlinnerjoin` function automatically detects the shared column between the tables. Specify the `toy_store` catalog and the `dbo` schema for both the left and right tables. Use the `'LeftCatalog'` and `'LeftSchema'` name-

value pair arguments for the left table, and the 'RightCatalog' and 'RightSchema' name-value pair arguments for the right table.

`data` is a table that contains the matched rows from the two tables.

```
lefttable = 'productTable';
righttable = 'suppliers';
data = sqlinnerjoin(conn,lefttable,righttable,'LeftCatalog','toy_store', ...
    'LeftSchema','dbo','RightCatalog','toy_store','RightSchema','dbo');
```

Display the first three rows of matched data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,3)
```

```
ans =
```

```
3×10 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0035e+05	1001	14	'Building Blocks'	1001
2	4.0031e+05	1002	9	'Painting Set'	1002
3	4.01e+05	1009	17	'Slinky'	1009

Close the database connection.

```
close(conn)
```

Specify Key for Joining Two Database Tables

Use an ODBC connection to import joined product data from two Microsoft® SQL Server® database tables into MATLAB®. Specify the key to use for joining the tables.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. Specify the key, or shared column, between the tables using the 'Keys' name-value pair argument.

`data` is a table that contains the matched rows from the two tables.

```
lefttable = 'productTable';
righttable = 'suppliers';
data = sqlinnerjoin(conn,lefttable,righttable,'Keys','supplierNumber');
```

Display the first three rows of matched data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,3)
```

```
ans =
```

```
3×10 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0035e+05	1001	14	'Building Blocks'	1001
2	4.0031e+05	1002	9	'Painting Set'	1002
3	4.01e+05	1009	17	'Slinky'	1009

Close the database connection.

```
close(conn)
```

Join Data Using Left and Right Keys

Use an ODBC connection to import employee data from an inner join between two Microsoft® SQL Server® database tables into MATLAB®. Specify the left and right keys for the join.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `employees` and `departments`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Join two database tables, `employees` and `departments`, to find the managers for particular departments. The `employees` table is the left table of the join, and the `departments` table is the right table of the join. Here, the column names of the keys are different. Specify the `MANAGER_ID` key in the left table using the `'LeftKeys'` name-value pair argument. Specify the `DEPT_MANAGER_ID` key in the right table using the `'RightKeys'` name-value pair argument.

`data` is a table that contains the matched rows from the two tables.

```
lefttable = 'employees';
righttable = 'departments';
data = sqlinnerjoin(conn,lefttable,righttable,'LeftKeys','MANAGER_ID', ...
    'RightKeys','DEPT_MANAGER_ID');
```

Display the first three rows of joined data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,3)
```

```
ans =
```

```
3×15 table
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE
101	'Neena'	'Kochhar'	'NKOCHHAR'	'515.123.4568'	'2005-09-21 00:00:00'
102	'Lex'	'De Haan'	'LDEHAAN'	'515.123.4569'	'2001-01-13 00:00:00'
104	'Bruce'	'Ernst'	'BERNST'	'590.423.4568'	'2007-05-21 00:00:00'

Close the database connection.

```
close(conn)
```

Limit Number of Rows in Joined Data

Use an ODBC connection to import joined product data from two Microsoft® SQL Server® database tables into MATLAB®. Specify the number of rows to return.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlinnerjoin` function automatically detects the shared column between the tables. Specify the number of rows to return using the `'MaxRows'` name-value pair argument.

```

lefttable = 'productTable';
righttable = 'suppliers';
data = sqlinnerjoin(conn,lefttable,righttable,'MaxRows',3)

```

```
data =
```

```
3×10 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0035e+05	1001	14	'Building Blocks'	1001
2	4.0031e+05	1002	9	'Painting Set'	1002
8	2.1257e+05	1001	5	'Train Set'	1001

`data` is a table that contains three of the matched rows from the two tables. The columns from the right table appear to the right of the columns from the left table.

Close the database connection.

```
close(conn)
```

Preserve Variable Names in Joined Data

Import joined product data from two Microsoft® SQL Server® database tables into MATLAB® by using an ODBC connection. One of the tables contains a variable name with a non-ASCII character. When importing data, preserve the names of all the variables.

Create an ODBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `productTable` and `suppliers`.

```

datasource = "MSSQLServerAuth";
conn = database(datasource, "", "");

```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Add a column to the database table `productTable`. The column name contains a non-ASCII character.

```

sqlquery = "ALTER TABLE productTable ADD tamaño varchar(30)";
execute(conn,sqlquery)

```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlinnerjoin` function automatically detects the shared column between the tables. Specify the number of rows to return using the `'MaxRows'` name-value pair argument.


```

lefttable = 'productTable';
righttable = 'suppliers';
data = sqlinnerjoin(conn,lefttable,righttable,'MaxRows',3)

```

```

data=3x11 table
  productNumber    stockNumber    supplierNumber    unitCost    productDescription    tama_
  _____
           1          4.0035e+05          1001             14    {'Building Blocks'}    {0x0 cha
           2          4.0031e+05          1002             9     {'Painting Set' }    {0x0 cha
           8          2.1257e+05          1001             5     {'Train Set' }      {0x0 cha

```

`data` is a table that contains three of the matched rows from the two tables. The `sqlinnerjoin` function converts the name of the new variable into ASCII characters.

Preserve the name of the variable that contains the non-ASCII character by specifying the `VariableNamingRule` name-value pair argument. Import the data again.

```

data = sqlinnerjoin(conn,lefttable,righttable,'MaxRows',3, ...
    'VariableNamingRule','preserve')

```

```

data=3x11 table
  productNumber    stockNumber    supplierNumber    unitCost    productDescription    tamaño
  _____
           1          4.0035e+05          1001             14    {'Building Blocks'}    {0x0 cha
           2          4.0031e+05          1002             9     {'Painting Set' }    {0x0 cha
           8          2.1257e+05          1001             5     {'Train Set' }      {0x0 cha

```

The `sqlinnerjoin` function preserves the non-ASCII character in the variable name.

Close the database connection.

```
close(conn)
```

Filter Rows in Joined Data

Use an ODBC connection to import product data from an inner join between two Microsoft® SQL Server® database tables into MATLAB®. Specify the row filter condition to use for joining the tables.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank username and password. The database contains the tables `productTable` and `suppliers`.

```

datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');

```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Join the two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlinnerjoin` function automatically detects the shared column between the tables. The table `data` contains the matched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlinnerjoin(conn, lefttable, righttable);
```

Display the first five rows of matched data. The columns from the right table appear to the right of the columns from the left table.

```
head(data, 5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0034e+05	1001	14	{'Building Blocks' }	100
2	4.0031e+05	1002	9	{'Painting Set' }	100
3	4.01e+05	1009	17	{'Slinky' }	100
4	4.0034e+05	1008	21	{'Space Cruiser' }	100
5	4.0046e+05	1005	3	{'Tin Soldier' }	100

Join the same tables, but this time use a row filter. The filter condition is that `unitCost` must be less than 15. Again, display the first five rows of matched data.

```
rf = rowfilter("unitCost");
rf = rf.unitCost < 15;
data = sqlinnerjoin(conn, lefttable, righttable, "RowFilter", rf);
head(data, 5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0034e+05	1001	14	{'Building Blocks' }	100
2	4.0031e+05	1002	9	{'Painting Set' }	100
5	4.0046e+05	1005	3	{'Tin Soldier' }	100
6	4.0088e+05	1004	8	{'Sail Boat' }	100
8	2.1257e+05	1001	5	{'Train Set' }	100

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

lefttable — Left table

character vector | string scalar

Left table, specified as a character vector or string scalar. Specify the name of the database table on the left side of the join.

Example: `'inventoryTable'`

Data Types: `char` | `string`

righttable – Right table

character vector | string scalar

Right table, specified as a character vector or string scalar. Specify the name of the database table on the right side of the join.

Example: 'productTable'

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `data = sqlinnerjoin(conn, lefttable, righttable, 'LeftCatalog', 'toy_store', 'LeftSchema', 'dbo', 'RightCatalog', 'toy_shop', 'RightSchema', 'toys', 'MaxRows', 5)` performs an inner join between left and right tables by specifying the catalog and schema for both tables and returns five matched rows.

LeftCatalog – Left catalog

character vector | string scalar

Left catalog, specified as the comma-separated pair consisting of 'LeftCatalog' and a character vector or string scalar. Specify the database catalog name where the left table of the join is stored.

Example: 'LeftCatalog', 'toy_store'

Data Types: char | string

RightCatalog – Right catalog

character vector | string scalar

Right catalog, specified as the comma-separated pair consisting of 'RightCatalog' and a character vector or string scalar. Specify the database catalog name where the right table of the join is stored.

Example: 'RightCatalog', 'toy_store'

Data Types: char | string

LeftSchema – Left schema

character vector | string scalar

Left schema, specified as the comma-separated pair consisting of 'LeftSchema' and a character vector or string scalar. Specify the database schema name where the left table of the join is stored.

Example: 'LeftSchema', 'dbo'

Data Types: char | string

RightSchema – Right schema

character vector | string scalar

Right schema, specified as the comma-separated pair consisting of 'RightSchema' and a character vector or string scalar. Specify the database schema name where the right table of the join is stored.

Example: 'RightSchema', 'dbo'

Data Types: char | string

Keys — Keys

character vector | string scalar | cell array of character vectors | string array

Keys, specified as the comma-separated pair consisting of 'Keys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. Use this name-value pair argument to identify the shared keys (columns) between the two tables to join.

You cannot use this name-value pair argument with the 'LeftKeys' and 'RightKeys' name-value pair arguments.

Example: 'Keys', 'MANAGER_ID'

Data Types: char | string | cell

LeftKeys — Left keys

character vector | string scalar | cell array of character vectors | string array

Left keys, specified as the comma-separated pair consisting of 'LeftKeys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the left table for the join to the right table.

Use this name-value pair argument with the 'RightKeys' name-value pair argument. Both arguments must specify the same number of keys. The `sqlinnerjoin` function pairs the values of the keys based on their order.

Example: 'LeftKeys', ["productNumber" "Price"], 'RightKeys', ["productNumber" "Price"]

Data Types: char | string | cell

RightKeys — Right keys

character vector | string scalar | cell array of character vectors | string array

Right keys, specified as the comma-separated pair consisting of 'RightKeys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the right table for the join to the left table.

Use this name-value pair argument with the 'LeftKeys' name-value pair argument. Both arguments must specify the same number of keys. The `sqlinnerjoin` function pairs the values of the keys based on their order.

Example: 'LeftKeys', ["productIdentifier" "Cost"], 'RightKeys', ["productNumber" "Price"]

Data Types: char | string | cell

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of 'MaxRows' and a positive numeric scalar. By default, the `sqlinnerjoin` function returns all rows from the

executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: 'MaxRows', 10

Data Types: double

VariableNamingRule — Variable naming rule

"modify" (default) | "preserve"

Variable naming rule, specified as the comma-separated pair consisting of 'VariableNamingRule' and one of these values:

- "modify" — Remove non-ASCII characters from variable names when the `sqlinnerjoin` function imports data.
- "preserve" — Preserve most variable names when the `sqlinnerjoin` function imports data. For details, see the Limitations section.

Example: 'VariableNamingRule', "modify"

Data Types: string

RowFilter — Row filter condition

<unconstrained> (default) | `matlab.io.RowFilter` object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; sqlinnerjoin(conn, lefttable, righttable, "RowFilter", rf)`

Output Arguments

data — Joined data

table

Joined data, returned as a table that contains the matched rows from the join of the left and right tables. `data` also contains a variable for each column in the left and right tables.

For columns that have numeric data types in the database table, the variable data types in `data` are `double` by default. For columns that have text, `date`, `time`, or `timestamp` data types in the database table, the variable data types are cell arrays of character vectors by default.

If the column names are shared between the joined database tables and have the same case, then the `sqlinnerjoin` function adds a unique suffix to the corresponding variable names in `data`.

Limitations

The name-value argument `VariableNamingRule` has these limitations:

- The `sqlinnerjoin` function returns an error if you specify the `VariableNamingRule` name-value argument with the `SQLImportOptions` object `opts`.
- When the `VariableNamingRule` name-value argument is set to the value "modify":
 - The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the table data type.

- The length of each variable name must be less than the number returned by `nameLengthmax`.

Version History

Introduced in R2018a

R2023a: Selectively join data based on filter condition

You can use the `RowFilter` when joining data from database tables.

See Also

`sqlfind` | `sqlread` | `sqlouterjoin` | `database` | `close`

Topics

“Join Tables Using Command Line” on page 5-57

“Importing Data Common Errors” on page 3-3

sqlouterjoin

Package: database.odbcc

Outer join between two database tables

Syntax

```
data = sqlouterjoin(conn,lefttable,righttable)
data = sqlouterjoin(conn,lefttable,righttable,Name,Value)
```

Description

`data = sqlouterjoin(conn,lefttable,righttable)` returns a table resulting from an outer join between the left and right database tables. This function matches rows using all shared columns, or keys, in both database tables. The outer join retains the matched and unmatched rows between the two tables. Executing this function is the equivalent of writing the SQL statement `SELECT * FROM lefttable,righttable OUTER JOIN lefttable.key = righttable.key`.

`data = sqlouterjoin(conn,lefttable,righttable,Name,Value)` uses additional options specified by one or more name-value arguments. For example, specify `Keys = "productNumber"` to use the `productNumber` column as a key for joining the two database tables.

Examples

Join Two Database Tables

Use an ODBC connection to import product data from an outer join between two Microsoft® SQL Server® database tables into MATLAB®.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlouterjoin` function automatically detects the shared column between the tables.

`data` is a table that contains the matched and unmatched rows from the two tables.

```
lefttable = 'productTable';
righttable = 'suppliers';
data = sqlouterjoin(conn,lefttable,righttable);
```

Display the first three rows of joined data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,3)
```

```
ans =
```

```
3×10 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0035e+05	1001	14	'Building Blocks'	1001
2	4.0031e+05	1002	9	'Painting Set'	1002
3	4.01e+05	1009	17	'Slinky'	1009

Close the database connection.

```
close(conn)
```

Join Two Database Tables in Catalog and Schema

Use an ODBC connection to import product data from an outer join between two Microsoft® SQL Server® database tables into MATLAB®. Specify the database catalog and schema where the tables are stored.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlouterjoin` function automatically detects the shared column between the tables. Specify the `toy_store` catalog and the `dbo` schema for both the left and right tables. Use the `'LeftCatalog'` and `'LeftSchema'` name-

value pair arguments for the left table, and the 'RightCatalog' and 'RightSchema' name-value pair arguments for the right table.

`data` is a table that contains the matched and unmatched rows from the two tables.

```
lefttable = 'productTable';
righttable = 'suppliers';
data = sqlouterjoin(conn,lefttable,righttable,'LeftCatalog','toy_store', ...
    'LeftSchema','dbo','RightCatalog','toy_store','RightSchema','dbo');
```

Display the first three rows of joined data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,3)
```

```
ans =
```

```
3×10 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0035e+05	1001	14	'Building Blocks'	1001
2	4.0031e+05	1002	9	'Painting Set'	1002
3	4.01e+05	1009	17	'Slinky'	1009

Close the database connection.

```
close(conn)
```

Specify Key for Joining Two Database Tables

Use an ODBC connection to import joined product data from two Microsoft® SQL Server® database tables into MATLAB®. Specify the key to use for joining the tables.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. Specify the key, or shared column, between the tables using the 'Keys' name-value pair argument.

`data` is a table that contains the matched and unmatched rows from the two tables.

```
lefttable = 'productTable';
righttable = 'suppliers';
data = sqlouterjoin(conn,lefttable,righttable,'Keys','supplierNumber');
```

Display the first three rows of joined data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,3)
```

```
ans =
```

```
3×10 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0035e+05	1001	14	'Building Blocks'	1001
2	4.0031e+05	1002	9	'Painting Set'	1002
3	4.01e+05	1009	17	'Slinky'	1009

Close the database connection.

```
close(conn)
```

Join Data Using Left and Right Keys

Use an ODBC connection to import employee data from an outer join between two Microsoft® SQL Server® database tables into MATLAB®. Specify the left and right keys for the join.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `employees` and `departments`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Join two database tables, `employees` and `departments`, to find the managers for particular departments. The `employees` table is the left table of the join, and the `departments` table is the right table of the join. Here, the column names of the keys are different. Specify the `MANAGER_ID` key in the left table using the `'LeftKeys'` name-value pair argument. Specify the `DEPT_MANAGER_ID` key in the right table using the `'RightKeys'` name-value pair argument.

`data` is a table that contains the matched and unmatched rows from the two tables.

```
lefttable = 'employees';
righttable = 'departments';
data = sqlouterjoin(conn,lefttable,righttable,'LeftKeys','MANAGER_ID', ...
    'RightKeys','DEPT_MANAGER_ID');
```

Display the last three unmatched rows of joined data. Display the last five variables of the joined data.

```
tail(data(:,end-4:end),3)
```

```
ans =
```

```
3x5 table
```

DEPARTMENT_ID	DEPARTMENT_ID_1	DEPARTMENT_NAME	DEPT_MANAGER_ID	LOCATION_ID
NaN	230	'IT Helpdesk'	NaN	1700
NaN	40	'Human Resources'	203	2400
NaN	10	'Administration'	200	1700

Close the database connection.

```
close(conn)
```

Create Right Join Using Left and Right Keys

Use an ODBC connection to import joined employee data from two Microsoft® SQL Server® database tables into MATLAB®. Create a right join and specify the left and right keys for the join.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `employees` and `departments`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Join two database tables, `employees` and `departments`, to find the managers for particular departments. The `employees` table is the left table of the join, and the `departments` table is the right table of the join. Here, the column names of the keys are different. Specify the `MANAGER_ID` key in the left table using the `'LeftKeys'` name-value pair argument. Specify the `DEPT_MANAGER_ID` key in the right table using the `'RightKeys'` name-value pair argument. Create a right join using the `'Type'` name-value pair argument.

```

lefttable = 'employees';
righttable = 'departments';
data = sqlouterjoin(conn,lefttable,righttable,'LeftKeys','MANAGER_ID', ...
    'RightKeys','DEPT_MANAGER_ID','Type','right');

```

`data` is a table that contains the matched rows from the two tables and the unmatched rows from the right table only.

Display the last three unmatched rows of joined data. Display the last five variables of the joined data.

```
tail(data(:,end-4:end),3)
```

```
ans =
```

```
3x5 table
```

DEPARTMENT_ID	DEPARTMENT_ID_1	DEPARTMENT_NAME	DEPT_MANAGER_ID	LOCATION_ID
NaN	250	'Retail Sales'	NaN	1700
NaN	260	'Recruiting'	NaN	1700
NaN	270	'Payroll'	NaN	1700

Close the database connection.

```
close(conn)
```

Limit Number of Rows in Joined Data

Use an ODBC connection to import joined product data from two Microsoft® SQL Server® database tables into MATLAB®. Specify the number of rows to return.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `productTable` and `suppliers`.

```

datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');

```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlouterjoin` function automatically detects the shared column between the tables. Specify the number of rows to return using the `'MaxRows'` name-value pair argument.

```

lefttable = 'productTable';
righttable = 'suppliers';
data = sqlouterjoin(conn,lefttable,righttable,'MaxRows',3)

```

```
data =
```

```
3×10 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
7	3.8912e+05	1007	16	'Engine Kit'	1007
8	2.1257e+05	1001	5	'Train Set'	1001
9	1.2597e+05	1003	13	'Victorian Doll'	1003

`data` is a table that contains three of the matched and unmatched rows from the two tables. The columns from the right table appear to the right of the columns from the left table.

Close the database connection.

```
close(conn)
```

Preserve Variable Names in Joined Data

Import joined product data from two Microsoft® SQL Server® database tables into MATLAB® by using an ODBC connection. One of the tables contains a variable name with a non-ASCII character. When importing data, preserve the names of all the variables.

Create an ODBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the tables `productTable` and `suppliers`.

```

datasource = "MSSQLServerAuth";
conn = database(datasource, "", "");

```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Add a column to the database table `productTable`. The column name contains a non-ASCII character.

```

sqlquery = "ALTER TABLE productTable ADD tamaño varchar(30)";
execute(conn,sqlquery)

```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlouterjoin` function automatically detects the shared column between the tables. Specify the number of rows to return using the `'MaxRows'` name-value pair argument.

```

lefttable = 'productTable';
righttable = 'suppliers';
data = sqlouterjoin(conn,lefttable,righttable,'MaxRows',3)

```

```

data=3x11 table
  productNumber    stockNumber    supplierNumber    unitCost    productDescription    tama_o
  _____    _____    _____    _____    _____    _____
           7           3.8912e+05           1007           16    {'Engine Kit'   }    {0x0 cha
           8           2.1257e+05           1001           5     {'Train Set'   }    {0x0 cha
           9           1.2597e+05           1003           13    {'Victorian Doll'}    {0x0 cha

```

`data` is a table that contains three of the matched rows from the two tables. The `sqlouterjoin` function converts the name of the new variable into ASCII characters.

Preserve the name of the variable that contains the non-ASCII character by specifying the `VariableNamingRule` name-value pair argument. Import the data again.

```

data = sqlouterjoin(conn,lefttable,righttable,'MaxRows',3, ...
    'VariableNamingRule','preserve')

```

```

data=3x11 table
  productNumber    stockNumber    supplierNumber    unitCost    productDescription    tamaño
  _____    _____    _____    _____    _____    _____
           7           3.8912e+05           1007           16    {'Engine Kit'   }    {0x0 cha
           8           2.1257e+05           1001           5     {'Train Set'   }    {0x0 cha
           9           1.2597e+05           1003           13    {'Victorian Doll'}    {0x0 cha

```

The `sqlouterjoin` function preserves the non-ASCII character in the variable name.

Close the database connection.

```
close(conn)
```

Filter Rows in Joined Data

Use an ODBC connection to import product data from an outer join between two Microsoft® SQL Server® database tables into MATLAB®. Specify the row filter condition to use for joining the tables.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank username and password. The database contains the tables `productTable` and `suppliers`.

```

datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');

```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Join the two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlouterjoin` function automatically detects the shared column between the tables. The table data contains the matched and unmatched rows from the two tables..

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlouterjoin(conn, lefttable, righttable);
```

Display the first five rows of matched data. The columns from the right table appear to the right of the columns from the left table.

```
head(data, 5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0034e+05	1001	14	{'Building Blocks' }	100
2	4.0031e+05	1002	9	{'Painting Set' }	100
3	4.01e+05	1009	17	{'Slinky' }	100
4	4.0034e+05	1008	21	{'Space Cruiser' }	100
5	4.0046e+05	1005	3	{'Tin Soldier' }	100

Join the same tables, but this time use a row filter. The filter condition is that `unitCost` must be less than 15. Again, display the first five rows of matched data.

```
rf = rowfilter("unitCost");
rf = rf.unitCost < 15;
data = sqlouterjoin(conn, lefttable, righttable, "RowFilter", rf);
head(data, 5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0034e+05	1001	14	{'Building Blocks' }	100
2	4.0031e+05	1002	9	{'Painting Set' }	100
5	4.0046e+05	1005	3	{'Tin Soldier' }	100
6	4.0088e+05	1004	8	{'Sail Boat' }	100
8	2.1257e+05	1001	5	{'Train Set' }	100

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

lefttable — Left table

character vector | string scalar

Left table, specified as a character vector or string scalar. Specify the name of the database table on the left side of the join.

Example: 'inventoryTable'

Data Types: char | string

righttable — Right table

character vector | string scalar

Right table, specified as a character vector or string scalar. Specify the name of the database table on the right side of the join.

Example: 'productTable'

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: data =

`sqlouterjoin(conn, lefttable, righttable, 'Type', 'left', 'MaxRows', 5)` performs an outer left join between left and right tables and returns five rows of the joined data.

LeftCatalog — Left catalog

character vector | string scalar

Left catalog, specified as the comma-separated pair consisting of 'LeftCatalog' and a character vector or string scalar. Specify the database catalog name where the left table of the join is stored.

Example: 'LeftCatalog', 'toy_store'

Data Types: char | string

RightCatalog — Right catalog

character vector | string scalar

Right catalog, specified as the comma-separated pair consisting of 'RightCatalog' and a character vector or string scalar. Specify the database catalog name where the right table of the join is stored.

Example: 'RightCatalog', 'toy_store'

Data Types: char | string

LeftSchema — Left schema

character vector | string scalar

Left schema, specified as the comma-separated pair consisting of 'LeftSchema' and a character vector or string scalar. Specify the database schema name where the left table of the join is stored.

Example: 'LeftSchema', 'dbo'

Data Types: char | string

RightSchema — Right schema

character vector | string scalar

Right schema, specified as the comma-separated pair consisting of 'RightSchema' and a character vector or string scalar. Specify the database schema name where the right table of the join is stored.

Example: 'RightSchema', 'dbo'

Data Types: char | string

Keys — Keys

character vector | string scalar | cell array of character vectors | string array

Keys, specified as the comma-separated pair consisting of 'Keys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. Use this name-value pair argument to identify the shared keys (columns) between the two tables to join.

You cannot use this name-value pair argument with the 'LeftKeys' and 'RightKeys' name-value pair arguments.

Example: 'Keys', 'MANAGER_ID'

Data Types: char | string | cell

LeftKeys — Left keys

character vector | string scalar | cell array of character vectors | string array

Left keys, specified as the comma-separated pair consisting of 'LeftKeys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the left table for the join to the right table.

Use this name-value pair argument with the 'RightKeys' name-value pair argument. Both arguments must specify the same number of keys. The `sqlouterjoin` function pairs the values of the keys based on their order.

Example: 'LeftKeys', ["productNumber" "Price"], 'RightKeys', ["productNumber" "Price"]

Data Types: char | string | cell

RightKeys — Right keys

character vector | string scalar | cell array of character vectors | string array

Right keys, specified as the comma-separated pair consisting of 'RightKeys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the right table for the join to the left table.

Use this name-value pair argument with the 'LeftKeys' name-value pair argument. Both arguments must specify the same number of keys. The `sqlouterjoin` function pairs the values of the keys based on their order.

Example: 'LeftKeys', ["productIdentifier" "Cost"], 'RightKeys', ["productNumber" "Price"]

Data Types: char | string | cell

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of 'MaxRows' and a positive numeric scalar. By default, the `sqlouterjoin` function returns all rows from the

executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: `'MaxRows', 10`

Data Types: `double`

Type — Outer join type

`'full' (default) | 'left' | 'right'`

Outer join type, specified as the comma-separated pair consisting of `'Type'` and one of these values:

- `'full'` — A full join retrieves records that have matching values in the selected column of both tables, and unmatched records from both the left and right tables.
- `'left'` — A left join retrieves records that have matching values in the selected column of both tables, and unmatched records from the left table only.
- `'right'` — A right join retrieves records that have matching values in the selected column of both tables, and unmatched records from the right table only.

You can specify these values as a character vector or string scalar.

Not all databases support all join types. For an unsupported database, you must use the `sqlread` function to import data from both tables into MATLAB. Then, use the `outerjoin` function to join tables in the MATLAB workspace.

Example: `'Type', 'left'`

VariableNamingRule — Variable naming rule

`"modify" (default) | "preserve"`

Variable naming rule, specified as the comma-separated pair consisting of `'VariableNamingRule'` and one of these values:

- `"modify"` — Remove non-ASCII characters from variable names when the `sqlouterjoin` function imports data.
- `"preserve"` — Preserve most variable names when the `sqlouterjoin` function imports data. For details, see the Limitations section.

Example: `'VariableNamingRule', "modify"`

Data Types: `string`

RowFilter — Row filter condition

`<unconstrained> (default) | matlab.io.RowFilter object`

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; sqlouterjoin(conn, lefttable, righttable, "RowFilter", rf)`

Output Arguments

data — Joined data

`table`

Joined data, returned as a table that contains rows matched by keys in the left and right database tables and the retained unmatched rows. `data` also contains a variable for each column in the left and right tables.

By default, the variable data types are `double` for columns that have `numeric` data types in the database table. For any `text`, `date`, `time`, or `timestamp` data types in the database table, the variable data type is a cell array of character vectors by default.

If the column names are shared between the joined database tables and have the same case, then the `outerjoin` function adds a unique suffix to the corresponding variable names in `data`.

The variables in `data` that correspond to columns in the left table contain `NULL` values when no matched rows exist in the right database table. Similarly, the variables that correspond to columns in the right table contain `NULL` values when no matched rows exist in the left database table.

Limitations

The name-value argument `VariableNamingRule` has these limitations:

- The `sqlouterjoin` function returns an error if you specify the `VariableNamingRule` name-value argument with the `SQLImportOptions` object `opts`.
- When the `VariableNamingRule` name-value argument is set to the value "modify":
 - The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the `table` data type.
 - The length of each variable name must be less than the number returned by `namelengthmax`.

Version History

Introduced in R2018a

R2023a: Selectively join data based on filter condition

You can use the `RowFilter` when joining data from database tables.

See Also

`sqlfind` | `sqlread` | `sqlinnerjoin` | `database` | `close`

Topics

"Join Tables Using Command Line" on page 5-57

"Importing Data Common Errors" on page 3-3

sqlread

Package: database.odbcc

Import data into MATLAB from database table

Syntax

```
data = sqlread(conn,tablename)
data = sqlread(conn,tablename,opts)
data = sqlread( ___,Name,Value)
[data,metadata] = sqlread( ___ )
```

Description

`data = sqlread(conn,tablename)` returns a table by importing data into MATLAB from a database table. Executing this function is the equivalent of writing a `SELECT * FROM tablename` SQL statement in ANSI SQL.

`data = sqlread(conn,tablename,opts)` customizes options for importing data from a database table using the `SQLImportOptions` object.

`data = sqlread(___,Name,Value)` specifies additional options using one or more name-value arguments with any of the previous input argument combinations. For example, specify `Catalog = "cat"` to import data from a database table stored in the "cat" catalog.

`[data,metadata] = sqlread(___)` also returns the `metadata` table, which contains metadata information about the imported data.

Examples

Import Data from Database Table

Use an ODBC connection to import product data from a database table into MATLAB® using a Microsoft® SQL Server® database. Then, perform a simple data analysis.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank username and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Import data from the database table `productTable`. The `sqlread` function returns a MATLAB® table that contains the product data.

```
tablename = 'productTable';
data = sqlread(conn,tablename);
```

Display the first five products.

```
head(data,5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	1.2597e+05	1003	13	{'Victorian Doll' }
8	2.1257e+05	1001	5	{'Train Set' }
7	3.8912e+05	1007	16	{'Engine Kit' }
2	4.0031e+05	1002	9	{'Painting Set' }
4	4.0034e+05	1008	21	{'Space Cruiser' }

Now, import the data using a row filter. The filter condition is that `unitCost` must be less than 15.

```
rf = rowfilter("unitCost");
rf = rf.unitCost < 15;
data = sqlread(conn,tablename,"RowFilter",rf);
```

Again, display the first five products.

```
head(data,5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	1.2597e+05	1003	13	{'Victorian Doll' }
8	2.1257e+05	1001	5	{'Train Set' }
2	4.0031e+05	1002	9	{'Painting Set' }
1	4.0034e+05	1001	14	{'Building Blocks' }
5	4.0046e+05	1005	3	{'Tin Soldier' }

Close the database connection.

```
close(conn)
```

Import Data from Database Table Using Import Options

Customize import options when importing data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize import options for different database columns. Import data using the `sqlread` function.

This example uses the `patients.xls` file, which contains the columns `Gender`, `Location`, `SelfAssessedHealthStatus`, and `Smoker`. The example also uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank username and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the patients database table using the patient information.

```
tablename = 'patients';
sqlwrite(conn, tablename, patients)
```

Create an SQLImportOptions object using the patients database table and the databaseImportOptions function.

```
opts = databaseImportOptions(conn, tablename)
```

```
opts =
  SQLImportOptions with properties:

    ExcludeDuplicates: false
    VariableNamingRule: 'modify'

    VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    VariableTypes: {'char', 'char', 'double' ... and 7 more}
    SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    FillValues: {'', '', NaN ... and 7 more }
    RowFilter: <unconstrained>

    VariableOptions: Show all 10 VariableOptions
```

Display the current import options for the variables selected in the SelectedVariableNames property of the SQLImportOptions object.

```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts, vars)
```

```
varOpts =
  1x10 SQLVariableImportOptions array with properties:

    Variable Options:
           (1) | (2) | (3) | (4) | (5) | (6) | (7) |
    Name: 'LastName' | 'Gender' | 'Age' | 'Location' | 'Height' | 'Weight' | 'Smoker' |
    Type: 'char' | 'char' | 'double' | 'char' | 'double' | 'double' | 'double' |
    MissingRule: 'fill' | 'fill' | 'fill' | 'fill' | 'fill' | 'fill' | 'fill' |
    FillValue: '' | '' | NaN | '' | NaN | NaN | NaN |
```

To access sub-properties of each variable, use getoptions

Change the data types for the Gender, Location, SelfAssessedHealthStatus, and Smoker variables using the setoptions function. Because the Gender, Location, and SelfAssessedHealthStatus variables indicate a finite set of repeating values, change their data type to categorical. Because the Smoker variable stores the values 0 and 1, change its data type to logical. Then, display the updated import options.

```
opts = setoptions(opts,{'Gender','Location','SelfAssessedHealthStatus'}, ...
    'Type','categorical');
opts = setoptions(opts,'Smoker','Type','logical');
```

```
varOpts = getoptions(opts,{'Gender','Location','Smoker', ...
    'SelfAssessedHealthStatus'})
```

```
varOpts =
    1x4 SQLVariableImportOptions array with properties:
```

Variable Options:

	(1)	(2)	(3)	(4)
Name:	'Gender'	'Location'	'Smoker'	'SelfAssessedHealthStatus'
Type:	'categorical'	'categorical'	'logical'	'categorical'
MissingRule:	'fill'	'fill'	'fill'	'fill'
FillValue:	<undefined>	<undefined>	0	<undefined>

To access sub-properties of each variable, use `getoptions`

Import the `patients` database table using the `sqlread` function, and display the last eight rows of the table.

```
data = sqlread(conn,tablename,opts);
tail(data)
```

LastName	Gender	Age	Location	Height	Weight	Smoker
{'Foster' }	Female	30	St. Mary's Medical Center	70	124	false
{'Gonzales' }	Male	48	County General Hospital	71	174	false
{'Bryant' }	Female	48	County General Hospital	66	134	false
{'Alexander' }	Male	25	County General Hospital	69	171	true
{'Russell' }	Male	44	VA Hospital	69	188	true
{'Griffin' }	Male	49	County General Hospital	70	186	false
{'Diaz' }	Male	45	County General Hospital	68	172	true
{'Hayes' }	Male	48	County General Hospital	66	177	false

Display a summary of the imported data. The `sqlread` function applies the import options to the variables in the imported data.

```
summary(data)
```

Variables:

LastName: 100×1 cell array of character vectors

Gender: 100×1 categorical

Values:

Female	53
Male	47

Age: 100×1 double

Values:

Min	25
Median	39

Max 50

Location: 100×1 categorical

Values:

County General Hospital	39
St. Mary s Medical Center	24
VA Hospital	37

Height: 100×1 double

Values:

Min	60
Median	67
Max	72

Weight: 100×1 double

Values:

Min	111
Median	142.5
Max	202

Smoker: 100×1 logical

Values:

True	34
False	66

Systolic: 100×1 double

Values:

Min	109
Median	122
Max	138

Diastolic: 100×1 double

Values:

Min	68
Median	81.5
Max	99

SelfAssessedHealthStatus: 100×1 categorical

Values:

Excellent	34
Fair	15
Good	40
Poor	11

Now set the filter condition to import only data for patients older than 40 years and not taller than 68 inches.

```
opts.RowFilter = opts.RowFilter.Age > 40 & opts.RowFilter.Height <= 68

opts =
  SQLImportOptions with properties:

    ExcludeDuplicates: false
    VariableNamingRule: 'modify'

    VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    VariableTypes: {'char', 'categorical', 'double' ... and 7 more}
    SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    FillValues: {'', <undefined>, NaN ... and 7 more }
    RowFilter: Age > 40 & Height <= 68

    VariableOptions: Show all 10 VariableOptions
```

Again, import the patients database table using the `sqlread` function, and display a summary of the imported data.

```
data = sqlread(conn,tablename,opts);
summary(data)
```

Variables:

LastName: 24×1 cell array of character vectors

Gender: 24×1 categorical

Values:

Female	17
Male	7

Age: 24×1 double

Values:

Min	41
Median	45.5
Max	50

Location: 24×1 categorical

Values:

County General Hospital	13
St. Mary s Medical Center	5
VA Hospital	6

Height: 24×1 double

Values:

Min	62
-----	----

Median	66
Max	68

Weight: 24×1 double

Values:

Min	119
Median	137
Max	194

Smoker: 24×1 logical

Values:

True	8
False	16

Systolic: 24×1 double

Values:

Min	114
Median	121.5
Max	138

Diastolic: 24×1 double

Values:

Min	68
Median	81.5
Max	96

SelfAssessedHealthStatus: 24×1 categorical

Values:

Excellent	7
Fair	3
Good	10
Poor	4

Delete the patients database table using the execute function.

```
sqlquery = ['DROP TABLE ' tablename];  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Import Data from Database Table in Specific Schema

Use an ODBC connection to import product data from a database table into MATLAB® using a Microsoft® SQL Server® database. Specify the schema where the database table is stored. Then, sort and filter the rows in the imported data and perform a simple data analysis.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Import data from the table `productTable`. Specify the database schema `dbo`. The data table contains the product data.

```
tablename = 'productTable';
data = sqlread(conn, tablename, 'Schema', 'dbo');
```

Display the first few products.

```
data(1:3, :)
```

```
ans =
```

```
3x5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	1.2597e+05	1003	13	'Victorian Doll'
8	2.1257e+05	1001	5	'Train Set'
7	3.8912e+05	1007	16	'Engine Kit'

Display the first few product descriptions.

```
data.productDescription(1:3)
```

```
ans =
```

```
3x1 cell array
```

```
{'Victorian Doll'}
{'Train Set'      }
{'Engine Kit'     }
```

Sort the rows in `data` by the product description column in alphabetical order.

```
column = 'productDescription';  
data = sortrows(data,column);
```

Display the first few product descriptions after sorting.

```
data.productDescription(1:3)
```

```
ans =
```

```
3×1 cell array  
  
{'Building Blocks'}  
{'Convertible'     }  
{'Engine Kit'      }
```

Close the database connection.

```
close(conn)
```

Import Specific Number of Rows from Database Table

Use an ODBC connection to import product data from a database table into MATLAB® using a Microsoft® SQL Server® database. Specify the maximum number of rows to import from the database table.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Import data from the table `productTable`. Import only three rows of data from the database table. The `data` table contains the product data.

```
tablename = 'productTable';  
data = sqlread(conn,tablename, 'MaxRows',3)
```

```
data =
```

```
3×5 table
```

<u>productNumber</u>	<u>stockNumber</u>	<u>supplierNumber</u>	<u>unitCost</u>	<u>productDescription</u>
9	1.2597e+05	1003	13	'Victorian Doll'
8	2.1257e+05	1001	5	'Train Set'
7	3.8912e+05	1007	16	'Engine Kit'

Close the database connection.

```
close(conn)
```

Preserve Variable Names When Importing Data

Import product data from a Microsoft® SQL Server® database table into MATLAB® by using an ODBC connection. The table contains a variable name with a non-ASCII character. When importing data, preserve the names of all the variables.

Create an ODBC database connection to an SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = "MSSQLServerAuth";
conn = database(datasource, "", "");
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Add a column to the database table `productTable`. The column name contains a non-ASCII character.

```
sqlquery = "ALTER TABLE productTable ADD tamaño varchar(30)";
execute(conn, sqlquery)
```

Import data from the database table `productTable`. The `sqlread` function returns a MATLAB table that contains the product data. Display the first three rows of the data in the table.

```
tablename = "productTable";
data = sqlread(conn, tablename);
head(data, 3)
```

```
ans=3×6 table
```

<u>productNumber</u>	<u>stockNumber</u>	<u>supplierNumber</u>	<u>unitCost</u>	<u>productDescription</u>	<u>tama_o</u>
9	1.2597e+05	1003	13	{'Victorian Doll'}	{0×0 cha
8	2.1257e+05	1001	5	{'Train Set' }	{0×0 cha
7	3.8912e+05	1007	16	{'Engine Kit' }	{0×0 cha

The `sqlread` function converts the name of the new variable into ASCII characters.

Preserve the name of the variable that contains the non-ASCII character by specifying the `VariableNamingRule` name-value pair argument. Import the data again.

```
data = sqlread(conn,tablename, ...
    'VariableNamingRule','preserve');
head(data,3)
```

```
ans=3x6 table
    productNumber    stockNumber    supplierNumber    unitCost    productDescription    tamaño
    _____    _____    _____    _____    _____    _____
            9      1.2597e+05      1003            13      {'Victorian Doll'}    {0x0 cha
            8      2.1257e+05      1001             5      {'Train Set'   }    {0x0 cha
            7      3.8912e+05      1007            16      {'Engine Kit'   }    {0x0 cha
```

The `sqlread` function preserves the non-ASCII character in the variable name.

Close the database connection.

```
close(conn)
```

Retrieve Metadata Information About Imported Data

Retrieve metadata information when importing data from a database table. Import data using the `sqlread` function and explore the metadata information by using dot notation.

This example uses the `outages.csv` file, which contains outage data. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MS SQL Server Auth";
conn = database(datasource, "", "");
```

Load outage information into the MATLAB® workspace.

```
outages = readtable("outages.csv");
```

Create the `outages` database table using the outage information.

```
tablename = "outages";
sqlwrite(conn,tablename,outages)
```

Import the data into the MATLAB workspace and return metadata information about the imported data.

```
[data,metadata] = sqlread(conn,tablename);
```

View the names of the variables in the imported data.

```
metadata.Properties.RowNames
```

```
ans = 6x1 cell array
    {'Region'   }
```

```

{'OutageTime'      }
{'Loss'            }
{'Customers'       }
{'RestorationTime'}
{'Cause'           }

```

View the data type of each variable in the imported data.

```
metadata.VariableType
```

```

ans = 6x1 cell array
    {'char' }
    {'char' }
    {'double'}
    {'double'}
    {'char' }
    {'char' }

```

View the missing data value for each variable in the imported data.

```
metadata.FillValue
```

```

ans = 6x1 cell array
    {0x0 char}
    {0x0 char}
    {[ NaN]}
    {[ NaN]}
    {0x0 char}
    {0x0 char}

```

View the indices of the missing data for each variable in the imported data.

```
metadata.MissingRows
```

```

ans = 6x1 cell array
    { 0x1 double}
    { 0x1 double}
    {604x1 double}
    {328x1 double}
    { 29x1 double}
    { 0x1 double}

```

Display the first eight rows of the imported data that contain missing restoration time. `data` contains restoration time in the fifth variable. Use the numeric indices to find the rows with missing data.

```

index = metadata.MissingRows{5,1};
nullrestoration = data(index,:);
head(nullrestoration)

```

```

ans=8x6 table
      Region      OutageTime      Loss      Customers      RestorationTime
-----
'SouthEast'  '2003-01-23 00:49:00.000'  530.14  2.1204e+05      ''
'NorthEast'  '2004-09-18 05:54:00.000'      0      0      ''

```

'MidWest'	'2002-04-20 16:46:00.000'	23141	NaN	''	'unkn
'NorthEast'	'2004-09-16 19:42:00.000'	4718	NaN	''	'unkn
'SouthEast'	'2005-09-14 15:45:00.000'	1839.2	3.4144e+05	''	'seve
'SouthEast'	'2004-08-17 17:34:00.000'	624.1	1.7879e+05	''	'seve
'SouthEast'	'2006-01-28 23:13:00.000'	498.78	NaN	''	'energ
'West'	'2003-06-20 18:22:00.000'	0	0	''	'energ

Delete the `outages` database table using the `execute` function.

```
sqlstr = "DROP TABLE ";
sqlquery = strcat(sqlstr,tablename);
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

opts — Database import options

SQLImportOptions object

Database import options, specified as an `SQLImportOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `data = sqlread(conn, 'inventoryTable', 'Catalog', 'toy_store', 'Schema', 'dbo', 'MaxRows', 5)` imports five rows of data from the database table `inventoryTable` stored in the `toy_store` catalog and the `dbo` schema.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: `string` | `char`

Schema — Database schema name

`string` scalar | `character` vector

Database schema name, specified as a string scalar or character vector. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Example: `Schema = "dbo"`

Data Types: `string` | `char`

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of `'MaxRows'` and a positive numeric scalar. By default, the `sqlread` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: `'MaxRows', 10`

Data Types: `double`

VariableNamingRule — Variable naming rule

`"modify"` (default) | `"preserve"`

Variable naming rule, specified as the comma-separated pair consisting of `'VariableNamingRule'` and one of these values:

- `"modify"` — Remove non-ASCII characters from variable names when the `sqlread` function imports data.
- `"preserve"` — Preserve most variable names when the `sqlread` function imports data. For details, see the Limitations section.

Example: `'VariableNamingRule', "modify"`

Data Types: `string`

RowFilter — Row filter condition

`<unconstrained>` (default) | `matlab.io.RowFilter` object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; sqlread(conn, tablename, "RowFilter", rf)`

Output Arguments

data — Imported data

`table`

Imported data, returned as a table. The rows of the table correspond to the rows in the database table `tablename`. The variables in the table correspond to each column in the database table. For columns that have numeric data types in the database table, the variable data types in `data` are `double` by default. For columns that have text, `date`, `time`, or `timestamp` data types in the database table, the variable data types are cell arrays of character vectors by default.

If the database table contains no data to import, then `data` is an empty table.

metadata — Metadata information

table

Metadata information, returned as a table with these variables.

Variable Name	Variable Description	Variable Data Type
<code>VariableType</code>	Data type of each variable in the imported data	Cell array of character vectors
<code>FillValue</code>	Value of missing data for each variable in the imported data	Cell array of missing data values
<code>MissingRows</code>	Indices for each occurrence of missing data in each variable of the imported data	Cell array of numeric indices

By default, the `sqlread` function imports text data as a character vector and numeric data as a `double`. `FillValue` is an empty character array (for text data) or `NaN` (for numeric data) by default. To change the missing data value to another value, use the `SQLImportOptions` object.

The `RowNames` property of the `metadata` table contains the names of the variables in the imported data.

Limitations

- The `sqlread` function returns an error when you use the `VariableNamingRule` name-value argument with the `SQLImportOptions` object `opts`.
- When the `VariableNamingRule` name-value pair argument is set to the value "modify":
 - The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the table data type.
 - The length of each variable name must be less than the number returned by `namelengthmax`.
- The `sqlread` function returns an error if you specify the `RowFilter` name-value argument with the `SQLImportOptions` object `opts`. It is ambiguous which of the `RowFilter` object to use in this case, especially if the filter conditions are different.

Version History

Introduced in R2018a

R2023a: Selectively import rows of data based on filter condition

You can use the `RowFilter` name-value argument to selectively import rows of data from a database table.

See Also

Functions

sqlfind | select | fetch | sqlinnerjoin | sqlouterjoin | database | close | databaseImportOptions | setoptions | getoptions | reset | execute | sqlupdate

Topics

“Import Data from Database Table Using sqlread Function” on page 5-58

“Customize Options for Importing Data from Database into MATLAB” on page 5-67

“Data Import Memory Management” on page 5-20

“Importing Data Common Errors” on page 3-3

sqlupdate

Package: database.odbcc

Update rows in database table

Syntax

```
sqlupdate(conn,tablename,data,filter)
sqlupdate( ____,Name,Value)
```

Description

`sqlupdate(conn,tablename,data,filter)` updates rows in the database table (`tablename`) with the rows from the MATLAB table (`data`) based on filter conditions (`filter`).

`sqlupdate(____,Name,Value)` specifies additional options using one or more name-value arguments with any of the previous input argument combinations. For example, `Catalog = "cat"` updates data from a database table stored in the "cat" catalog.

Examples

Update Database Rows

Update database rows based on filter conditions specified with row filters.

This example uses the `patients.xls` file, which contains the columns `LastName`, `Gender`, `Age`, `Location`, `Height`, `Weight`, `Smoker`, `Systolic`, `Diastolic`, and `SelfAssessedHealthStatus`. The example also uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank username and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';
sqlwrite(conn,tablename,patients)
```

Use the SQL ALTER statement to add the column `HighRisk` to the table `patients`.

```
sqlquery = 'ALTER TABLE patients ADD HighRisk bit';
execute(conn,sqlquery)
```

Import the `patients` database table using the `sqlread` function, and return metadata information about the imported data.

```
[data,metadata] = sqlread(conn,tablename);
```

Display the first 10 rows of the table. In MATLAB, all the values in the `HighRisk` column appear as `false`.

```
head(data,10)
```

LastName	Gender	Age	Location	Height	Weight	Smoker
{'Smith' }	{'Male' }	38	{'County General Hospital' }	71	176	0
{'Johnson' }	{'Male' }	43	{'VA Hospital' }	69	163	0
{'Williams' }	{'Female' }	38	{'St. Mary's Medical Center' }	64	131	0
{'Jones' }	{'Female' }	40	{'VA Hospital' }	67	133	0
{'Brown' }	{'Female' }	49	{'County General Hospital' }	64	119	0
{'Davis' }	{'Female' }	46	{'St. Mary's Medical Center' }	68	142	0
{'Miller' }	{'Female' }	33	{'VA Hospital' }	64	142	0
{'Wilson' }	{'Male' }	40	{'VA Hospital' }	68	180	0
{'Moore' }	{'Male' }	28	{'St. Mary's Medical Center' }	68	183	0
{'Taylor' }	{'Female' }	31	{'County General Hospital' }	66	132	0

Displaying the metadata shows that the values are `NULL` (missing elements) in the database.

```
metadata
```

```
metadata=11x3 table
```

	VariableType	FillValue	MissingRows
LastName	{'char' }	{0x0 char}	{ 0x1 double}
Gender	{'char' }	{0x0 char}	{ 0x1 double}
Age	{'double' }	{[NaN]}	{ 0x1 double}
Location	{'char' }	{0x0 char}	{ 0x1 double}
Height	{'double' }	{[NaN]}	{ 0x1 double}
Weight	{'double' }	{[NaN]}	{ 0x1 double}
Smoker	{'double' }	{[NaN]}	{ 0x1 double}
Systolic	{'double' }	{[NaN]}	{ 0x1 double}
Diastolic	{'double' }	{[NaN]}	{ 0x1 double}
SelfAssessedHealthStatus	{'char' }	{0x0 char}	{ 0x1 double}
HighRisk	{'logical' }	{[0]}	{100x1 double}

Now, identify patients who are considered high risk for developing some hypothetical health issue based on their age and their smoker status. First, create a table containing the new data to write to the database. This table requires only 1 (`true`) and 0 (`false`) values.

```
t = table([1;0], "VariableNames", "HighRisk");
head(t)
```

HighRisk
1
0

Create a row filter using the filter condition that a patient must be older than 35 years and a smoker to be considered high-risk.

```
rf = rowfilter(["Age", "Smoker"]);
rf = rf.Age > 35 & rf.Smoker == 1
```

```
rf =
  RowFilter with constraints:
    Age > 35 & Smoker == 1
  VariableNames: Age, Smoker
```

Update the `HighRisk` column using this filter to set the values to 1 (true) and using the `~rf` value of the filter to set the value to 0 (false).

```
sqlupdate(conn, "patients", t, {rf; ~rf});
```

Again, import the `patients` database table using the `sqlread` function, and display the first 10 rows.

```
data = sqlread(conn, tablename);
head(data, 10)
```

LastName	Gender	Age	Location	Height	Weight	Smoker
{'Smith' }	{'Male' }	38	{'County General Hospital' }	71	176	1
{'Johnson' }	{'Male' }	43	{'VA Hospital' }	69	163	0
{'Williams' }	{'Female' }	38	{'St. Mary's Medical Center' }	64	131	0
{'Jones' }	{'Female' }	40	{'VA Hospital' }	67	133	0
{'Brown' }	{'Female' }	49	{'County General Hospital' }	64	119	0
{'Davis' }	{'Female' }	46	{'St. Mary's Medical Center' }	68	142	0
{'Miller' }	{'Female' }	33	{'VA Hospital' }	64	142	1
{'Wilson' }	{'Male' }	40	{'VA Hospital' }	68	180	0
{'Moore' }	{'Male' }	28	{'St. Mary's Medical Center' }	68	183	0
{'Taylor' }	{'Female' }	31	{'County General Hospital' }	66	132	0

Delete the `patients` database table using the `execute` function.

```
sqlquery = ['DROP TABLE ' tablename];
execute(conn, sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

data — Updated data

MATLAB table

Updated data, specified as a MATLAB table. The table can contain one or more rows with updated data. The names of the variables in the table must be a subset of the column names of the database table.

Example: `data = table([1;0], "VariableNames", "NewName")`

Data Types: table

filter — Row filter condition

matlab.io.RowFilter object | cell array of matlab.io.RowFilter objects

Row filter condition, specified as a `matlab.io.RowFilter` object or cell array of `matlab.io.RowFilter` objects. Filters determine which database rows `sqlupdate` must update with which data. If multiple database rows match a filter, `sqlupdate` updates them with the same data. If a single database row matches multiple filters, its final state matches the data corresponding to the last matching filter.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5;`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `sqlupdate(conn, 'inventoryTable', data, rf, Catalog = "toy_store", Schema = "dbo")` updates the database `inventoryTable` stored in the `toy_store` catalog and the `dbo` schema.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: string | char

Schema — Database schema name

string scalar | character vector

Database schema name, specified as a string scalar or character vector. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Example: `Schema = "dbo"`

Data Types: `string` | `char`

Version History

Introduced in R2023a

See Also

`sqlread` | `sqlfind` | `select` | `fetch` | `sqlinnerjoin` | `sqlouterjoin` | `database` | `close` | `databaseImportOptions` | `setoptions` | `getoptions` | `reset`

sqlupdate

Update rows in SQLite database table

Syntax

```
sqlupdate(conn,tablename,data,filter)
sqlupdate( ____,Name,Value)
```

Description

`sqlupdate(conn,tablename,data,filter)` updates rows in the SQLite database table (`tablename`) with the rows from the MATLAB table (`data`) based on filter conditions (`filter`).

`sqlupdate(____,Name,Value)` specifies additional options using one or more name-value arguments with any of the previous input argument combinations. For example, `Catalog = "cat"` updates data from a database table stored in the "cat" catalog.

Examples

Update Database Rows

Update rows in the database table in the SQLite database file based on filter conditions specified with row filters.

Create the SQLite connection to the existing SQLite database file `inventory.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = "inventory.db";
conn = sqlite(dbfile);
```

Import all the data from `productTable`. The `results` output argument contains the imported data as a table. Display the first 10 rows of the table.

```
tablename = "productTable";
results = sqlread(conn,tablename);
head(results,10)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	125970	1003	13	"Victorian Doll"
8	212569	1001	5	"Train Set"
7	389123	1007	16	"Engine Kit"
2	400314	1002	9	"Painting Set"
4	400339	1008	21	"Space Cruiser"
1	400345	1001	14	"Building Blocks"
5	400455	1005	3	"Tin Soldier"
6	400876	1004	8	"Sail Boat"
3	400999	1009	17	"Slinky"
10	888652	1006	24	"Teddy Bear"

Use the SQL ALTER statement to add the column Recall to the table.

```
sqlquery = strcat("ALTER TABLE productTable ADD COLUMN Recall INTEGER DEFAULT 0");
execute(conn,sqlquery)
```

Now, identify products being recalled. First, create a table containing the new data to write to the database. This table requires only 1 (true) and 0 (false) values.

```
t = table([1;0], "VariableNames", "Recall");
head(t)
```

```
Recall
-----
      1
      0
```

Create a row filter using the filter condition that a supplier number must be either 1001 or greater than 1005 and the stock number must be greater than 400,000.

```
rf = rowfilter(["supplierNumber", "stockNumber"]);
rf = (rf.supplierNumber == 1001 | rf.supplierNumber > 1005) & rf.stockNumber > 400000
```

```
rf =
```

```
RowFilter with constraints:
```

```
(supplierNumber == 1001 | supplierNumber > 1005) & stockNumber > 400000
```

```
VariableNames: supplierNumber, stockNumber
```

Update the Recall column using this filter to set the values to 1 (true) and using the ~rf value of the filter to set the value to 0 (false).

```
sqlupdate(conn, "productTable", t, {rf;~rf});
```

Again, import the data from productTable and display the first 10 rows.

```
results = sqlread(conn,tablename);
head(results,10)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Recall
9	125970	1003	13	"Victorian Doll"	0
8	212569	1001	5	"Train Set"	0
7	389123	1007	16	"Engine Kit"	0
2	400314	1002	9	"Painting Set"	0
4	400339	1008	21	"Space Cruiser"	1
1	400345	1001	14	"Building Blocks"	1
5	400455	1005	3	"Tin Soldier"	0
6	400876	1004	8	"Sail Boat"	0
3	400999	1009	17	"Slinky"	1
10	888652	1006	24	"Teddy Bear"	1

Use the SQL ALTER statement to remove the Recall column from the table.

```
sqlquery = strcat("ALTER TABLE productTable DROP COLUMN Recall");
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

data — Updated data

MATLAB table

Updated data, specified as a MATLAB table. The table can contain one or more rows with updated data. The names of the variables in the table must be a subset of the column names of the database table.

Example: `data = table([1;0], "VariableNames", "NewName")`

Data Types: table

filter — Row filter condition

matlab.io.RowFilter object | cell array of matlab.io.RowFilter objects

Row filter condition, specified as a `matlab.io.RowFilter` object or a cell array of `matlab.io.RowFilter` objects. Filters determine which database rows `sqlupdate` must update with which data. If multiple database rows match a filter, `sqlupdate` updates them with the same data. If a single database row matches multiple filters, its final state matches the data corresponding to the last matching filter.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5;`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `sqlupdate(conn, 'inventoryTable', data, rf, Catalog = "toy_store", Schema = "dbo")` updates the database `inventoryTable` stored in the `toy_store` catalog and the `dbo` schema.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: `string` | `char`

Schema — Database schema name

`string` scalar | `character` vector

Database schema name, specified as a string scalar or character vector. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Example: `Schema = "dbo"`

Data Types: `string` | `char`

Version History

Introduced in R2023a

See Also

`sqlite` | `fetch` | `sqlwrite` | `close` | `sqlread`

sqlread

Import data into MATLAB from SQLite database table

Syntax

```
data = sqlread(conn,tablename)
data = sqlread(conn,tablename,Name=Value)
```

Description

`data = sqlread(conn,tablename)` returns a table by importing data into MATLAB from a database table with the MATLAB interface to SQLite. Executing this function is the equivalent of writing a `SELECT * FROM tablename` SQL statement in ANSI SQL.

`data = sqlread(conn,tablename,Name=Value)` specifies additional options using one or more name-value arguments. For example, `MaxRows=5` imports five rows of data.

Examples

Import Data from Database Table in SQLite Database File

Import all rows of data from a database table in an SQLite database file into MATLAB. Determine the highest unit cost among products in the table. Then, use the `sqlread` function with a filter to import only the data for products with a unit cost less than 15.

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Import all the data from `productTable`. The `results` output argument contains the imported data as a table.

```
tablename = "productTable";
results = sqlread(conn,tablename)
```

results=15x5 table

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	125970	1003	13	"Victorian Doll"
8	212569	1001	5	"Train Set"
7	389123	1007	16	"Engine Kit"
2	400314	1002	9	"Painting Set"
4	400339	1008	21	"Space Cruiser"
1	400345	1001	14	"Building Blocks"
5	400455	1005	3	"Tin Soldier"
6	400876	1004	8	"Sail Boat"
3	400999	1009	17	"Slinky"

10	888652	1006	24	"Teddy Bear"
11	408143	1004	11	"Convertible"
12	210456	1010	22	"Hugsy"
13	470816	1012	16	"Pancakes"
14	510099	1011	19	"Shawl"
15	899752	1011	20	"Snacks"

Determine the highest unit cost of the products. Access unit cost data using the variable of the results table. data is a vector that contains numeric unit costs. Find the maximum unit cost.

```
data = results.unitCost;
max(data)
```

```
ans = int64
     24
```

Now, import the data using a row filter. The filter condition is that unitCost must be less than 15.

```
rf = rowfilter("unitCost");
rf = rf.unitCost < 15;
results = sqlread(conn,tablename,"RowFilter",rf)
```

```
results=7x5 table
   productNumber  stockNumber  supplierNumber  unitCost  productDescription
   _____  _____  _____  _____  _____
         9         125970         1003         13      "Victorian Doll"
         8         212569         1001          5      "Train Set"
         2         400314         1002          9      "Painting Set"
         1         400345         1001         14      "Building Blocks"
         5         400455         1005          3      "Tin Soldier"
         6         400876         1004          8      "Sail Boat"
        11         408143         1004         11      "Convertible"
```

Close the SQLite connection.

```
close(conn)
```

Import Specific Number of Rows from Database Table

Use the sqlread function of the MATLAB® interface to SQLite to import a limited number of rows of data into MATLAB from a database table in an SQLite database file.

Create the SQLite connection conn to the existing SQLite database file tutorial.db. The database file contains the table productTable. The SQLite connection is an sqlite object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Import data from the table productTable. Import only three rows of data from the database table. The data table contains the product data.

```
tablename = "productTable";
data = sqlread(conn,tablename,MaxRows=3)
```

```
data=3x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
  _____  _____  _____  _____  _____
           9          125970          1003          13      "Victorian Doll"
           8          212569          1001           5      "Train Set"
           7          389123          1007          16      "Engine Kit"
```

Close the SQLite connection.

```
close(conn)
```

Remove Non-ASCII Characters in Variable Names When Importing Data

Import product data from an SQLite database table into MATLAB® by using the MATLAB interface to SQLite. The table contains a variable name with a non-ASCII character. When importing data, remove non-ASCII characters from the names of all the variables.

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Rename the `unitCost` column in the database table `productTable` to `tamaño`. The column name contains a non-ASCII character.

```
sqlquery = "ALTER TABLE productTable RENAME COLUMN unitCost TO tamaño";
execute(conn,sqlquery)
```

Import data from the database table `productTable`. The `sqlread` function returns a MATLAB table that contains the product data. Display the first three rows of the data in the table.

```
tablename = "productTable";
data = sqlread(conn,tablename);
head(data,3)
```

```
  productNumber  stockNumber  supplierNumber  tamaño  productDescription
  _____  _____  _____  _____  _____
           9          125970          1003          13      "Victorian Doll"
           8          212569          1001           5      "Train Set"
           7          389123          1007          16      "Engine Kit"
```

The `sqlread` function preserves non-ASCII characters in the name of the variable by default.

Remove the non-ASCII character in the name of the variable by specifying the `VariableNamingRule` name-value argument. Import the data again.

```
data = sqlread(conn,tablename, ...
    VariableNamingRule="modify");
head(data,3)
```

```
  productNumber  stockNumber  supplierNumber  tama_o  productDescription
  _____  _____  _____  _____  _____
```

9	125970	1003	13	"Victorian Doll"
8	212569	1001	5	"Train Set"
7	389123	1007	16	"Engine Kit"

The `sqlread` function removes the non-ASCII character in the variable name.

Rename the `tamaño` column in the database table `productTable` back to `unitCost`.

```
sqlquery = "ALTER TABLE productTable RENAME COLUMN tamaño TO unitCost";  
execute(conn,sqlquery)
```

Close the SQLite connection.

```
close(conn)
```

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `data = sqlread(conn, "inventoryTable", MaxRows=5)` imports five rows of data from the database table `inventoryTable`.

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as a positive numeric scalar. By default, the `sqlread` function returns all rows from the executed SQL query. Use this name-value argument to limit the number of rows imported into MATLAB.

Example: `MaxRows=10`

Data Types: `double`

VariableNamingRule — Variable naming rule

"preserve" (default) | "modify"

Variable naming rule, specified as one of these values:

- "preserve" — Preserve most variable names when the `sqlread` function imports data.
- "modify" — Remove non-ASCII characters from variable names when the `sqlread` function imports data.

Example: `VariableNamingRule="modify"`

Data Types: `string`

RowFilter — Row filter condition

<unconstrained> (default) | `matlab.io.RowFilter` object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; sqlread(conn,tablename,"RowFilter",rf)`

Output Arguments

data — Result data

table

Result data, returned as a table. The result data contains all rows of data from the executed SQL statement.

The `sqlread` function converts SQLite data types to MATLAB data types and represents NULL values accordingly.

SQLite Data Type	MATLAB Data Type	MATLAB Null Value Representation
<ul style="list-style-type: none"> • REAL • DOUBLE • FLOAT • NUMERIC • INT • TINYINT • SMALLINT • MEDIUMINT • BIGINT 	double	double(NaN)
<ul style="list-style-type: none"> • CHAR • VARCHAR 	string	<missing>
<ul style="list-style-type: none"> • DATE • DATETIME 	string	<missing>
<ul style="list-style-type: none"> • BLOB 	<i>N</i> x 1 uint8 vector	0 x 1 uint8 vector
<ul style="list-style-type: none"> • BOOLEAN 	int64	Not available

Version History

Introduced in R2022a

R2023a: Selectively import rows of data based on filter condition

You can use the `RowFilter` name-value argument to selectively import rows of data from a database table.

See Also**Objects**

`sqlite`

Functions

`fetch` | `sqlwrite` | `close`

Topics

“Import Data Using MATLAB Interface to SQLite” on page 5-32

“Insert Data into SQLite Database Table” on page 5-36

sqlwrite

Package: database.odbcc

Insert MATLAB data into database table

Syntax

```
sqlwrite(conn,tablename,data)
sqlwrite(conn,tablename,data,Name,Value)
```

Description

`sqlwrite(conn,tablename,data)` inserts data from a MATLAB table into a database table. If the table exists in the database, this function appends the data in the MATLAB table as rows in the existing database table. If the table does not exist in the database, this function creates a table with the specified table name and then inserts the data as rows in the new table. This syntax is the equivalent of executing SQL statements that contain the `CREATE TABLE` and `INSERT INTO ANSI SQL` syntaxes.

`sqlwrite(conn,tablename,data,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `'Catalog','toy_store'` inserts data into a database table that is located in the database catalog named `toy_store`.

Examples

Append Data into Existing Table

Use an ODBC connection to append product data from a MATLAB® table into an existing table in a Microsoft® SQL Server® database.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

To view the existing database table `productTable` before appending data, import its contents into MATLAB and display the last few rows.

```
tablename = 'productTable';
rows = sqlread(conn,tablename);
tail(rows,3)
```

ans =

3×5 table

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	4.7082e+05	1012	17	'Pancakes'
14	5.101e+05	1011	19	'Shawl'
15	8.9975e+05	1011	20	'Snacks'

Create a MATLAB table that contains the data for one product.

```
data = table(30,500000,1000,25,"Rubik's Cube", ...
    'VariableNames',{'productNumber' 'stockNumber' ...
    'supplierNumber' 'unitCost' 'productDescription'});
```

Append the product data into the database table productTable.

```
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB again and display the last few rows. The results contain a new row for the inserted product.

```
rows = sqlread(conn,tablename);
tail(rows,4)
```

ans =

4×5 table

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	4.7082e+05	1012	17	'Pancakes'
14	5.101e+05	1011	19	'Shawl'
15	8.9975e+05	1011	20	'Snacks'
30	5e+05	1000	25	'Rubik's Cube'

Close the database connection.

```
close(conn)
```

Insert Data into New Table

Use an ODBC connection to insert product data from MATLAB® into a new table in a Microsoft® SQL Server® database.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the Message property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Create a MATLAB table that contains data for two products.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"],'VariableNames',{'productNumber' ...
    'stockNumber' 'supplierNumber' 'unitCost' 'productDescription'});
```

Insert the product data into a new database table toyTable.

```
tablename = 'toyTable';
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows =
```

```
 2×5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
30	5e+05	1000	25	'Rubik's Cube'
40	6e+05	2000	30	'Doll House'

Close the database connection.

```
close(conn)
```

Specify Column Types When Inserting Data into New Table

Use an ODBC connection to insert product data from MATLAB® into a new table in a Microsoft® SQL Server® database. Specify the data types of the columns in the new database table.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Create a MATLAB table that contains data for two products.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"], 'VariableNames',{'productNumber' ...
    'stockNumber' 'supplierNumber' 'unitCost' 'productDescription'});
```

Insert the product data into a new database table `toyTable`. Use the `'ColumnType'` name-value pair argument and a string array to specify the data types of all the columns in the database table.

```
tablename = 'toyTable';
coltypes = ["numeric" "numeric" "numeric" "numeric" "varchar(255)"];
sqlwrite(conn,tablename,data,'ColumnType',coltypes)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows =
```

```
 2×5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
30	5e+05	1000	25	'Rubik's Cube'
40	6e+05	2000	30	'Doll House'

Close the database connection.

```
close(conn)
```

Insert Cell Array into Table

Use an ODBC connection to insert product data from MATLAB® into a new table in a Microsoft® SQL Server® database. Insert data stored as a cell array into the new database table.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the Message property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Create a cell array that contains data for two products.

```
c = {30,500000,1000,25,"Rubik's Cube";40,600000,2000,30,"Doll House"};
```

Convert the cell array to a MATLAB table by specifying the column names.

```
colnames = {'productNumber' 'stockNumber' 'supplierNumber' 'unitCost' ...
            'productDescription'};
data = cell2table(c,'VariableNames',colnames);
```

Insert the product data into a new database table toyTable.

```
tablename = 'toyTable';
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows =
```

```
 2×5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
30	5e+05	1000	25	'Rubik's Cube'
40	6e+05	2000	30	'Doll House'

Close the database connection.

```
close(conn)
```

Insert Structure into Table

Use an ODBC connection to insert product data from MATLAB® into a new table in a Microsoft® SQL Server® database. Insert data stored as a structure into the new database table.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the Message property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
    []
```

Create a structure array that contains data for two products.

```
s(1).productNumber = 30;
s(1).stockNumber = 500000;
s(1).supplierNumber = 1000;
s(1).unitCost = 25;
s(1).productDescription = "Rubik's Cube";

s(2).productNumber = 40;
s(2).stockNumber = 600000;
s(2).supplierNumber = 2000;
s(2).unitCost = 30;
s(2).productDescription = "Doll House";
```

Convert the structure to a MATLAB table.

```
data = struct2table(s);
```

Insert the product data into a new database table toyTable.

```
tablename = 'toyTable';
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows =
```

```
2×5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
30	5e+05	1000	25	'Rubik's Cube'
40	6e+05	2000	30	'Doll House'

Close the database connection.

```
close(conn)
```


Insert Numeric Array into Table

Use an ODBC connection to insert sales volume data from MATLAB® into an existing table in a Microsoft® SQL Server® database. Insert data stored as a numeric array into the existing database table.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the `salesVolume` table.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Create a numeric array that contains monthly sales volume data for a specific stock number. Specify the column names for the existing database table `salesVolume`.

```
n = [100000 1000 0 2000 500 3000 450 600 700 750 1450 0 0];
colnames = {'StockNumber' 'January' 'February' 'March' 'April' 'May' ...
            'June' 'July' 'August' 'September' 'October' 'November' 'December'};
```

Convert the numeric array to a MATLAB table.

```
data = array2table(n, 'VariableNames', colnames);
```

Insert the sales volume data into the database table `salesVolume`.

```
tablename = 'salesVolume';
sqlwrite(conn, tablename, data)
```

Import the contents of the database table into MATLAB and display the last three rows. The results contain a new row for the inserted sales volume data.

```
rows = sqlread(conn, tablename);
tail(rows, 3)
```

```
ans =
```

```
 3×13 table
```

StockNumber	January	February	March	April	May	June	July	August	Sep
5.101e+05	235	1800	1040	900	750	700	400	350	
8.9975e+05	123	1700	823	701	689	621	545	421	
1e+05	1000	0	2000	500	3000	450	600	700	

Close the database connection.

```
close(conn)
```

Insert Date Number into Table

Use an ODBC connection to insert inventory data from MATLAB® into an existing table in a Microsoft® SQL Server® database. Insert a date stored as a date number into the existing database table.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the table `inventoryTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Create a numeric array that contains inventory data for a specific product, including the date number 731011. Specify the column names for the existing database table `inventoryTable`.

```
n = [25 1000 50 731011];
colnames = {'productNumber' 'Quantity' 'Price' 'inventoryDate'};
```

Convert the numeric array to a MATLAB table.

```
data = array2table(n, 'VariableNames', colnames);
```

Convert the date value in the inventory data to a `datetime` array. The `sqlwrite` function does not accept date numbers as a valid data type for insertion.

```
n = data.inventoryDate;
data.inventoryDate = datetime(n, 'ConvertFrom', 'datenum');
```

Import the contents of the database table `inventoryTable` into MATLAB and display the last few rows.

```
tablename = 'inventoryTable';
rows = sqlread(conn, tablename);
tail(rows, 3)
```

```
ans =
```

```
 3×4 table
```

productNumber	Quantity	Price	inventoryDate
_____	_____	_____	_____

```

11          567          11      '2012-09-11 00:30:24.000'
12          1278         22      '2010-10-29 18:17:47.000'
13          1700         17      '2009-05-24 10:58:59.000'

```

Insert the inventory data into the database table `inventoryTable`. Specify the schema where the table is stored by using the 'Schema' name-value pair argument.

```
sqlwrite(conn,tablename,data,'Schema','dbo')
```

Import the contents of the database table into MATLAB again and display the last few rows. The results contain a new row for the inserted inventory data.

```
rows = sqlread(conn,tablename);
tail(rows,4)
```

```
ans =
```

```
4x4 table
```

productNumber	Quantity	Price	inventoryDate
11	567	11	'2012-09-11 00:30:24.000'
12	1278	22	'2010-10-29 18:17:47.000'
13	1700	17	'2009-05-24 10:58:59.000'
25	1000	50	'2001-06-09 00:00:00.000'

Close the database connection.

```
close(conn)
```

Insert NULL Number into Table

Use an ODBC connection to insert sales volume data from MATLAB® into an existing table in a Microsoft® SQL Server® database. Insert NULL numbers into the existing database table.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the table `salesVolume`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
[]
```

Create a numeric array that contains monthly sales volume data for a specific stock number, and includes a NULL number. The value `Inf` indicates a NULL value. Specify the column names for the existing database table `salesVolume`.

```
n = [100000 Inf 0 2000 500 3000 450 600 700 750 1450 0 0];
colnames = {'StockNumber' 'January' 'February' 'March' 'April' 'May' ...
            'June' 'July' 'August' 'September' 'October' 'November' 'December'};
```

Convert the numeric array to a MATLAB table.

```
data = array2table(n, 'VariableNames', colnames);
```

Convert the `Inf` value in the January variable to `NaN`. The `sqlwrite` function does not accept `Inf` values as valid missing data for insertion.

```
data.January = NaN;
```

Import the contents of the database table `salesVolume` into MATLAB and display the last few rows.

```
tablename = 'salesVolume';
rows = sqlread(conn, tablename);
tail(rows, 3)
```

ans =

3×13 table

StockNumber	January	February	March	April	May	June	July	August	Sep
4.7082e+05	3100	9400	1540	1500	1350	1190	900	867	
5.101e+05	235	1800	1040	900	750	700	400	350	
8.9975e+05	123	1700	823	701	689	621	545	421	

Insert the sales volume data into the database table `salesVolume`.

```
sqlwrite(conn, tablename, data)
```

Import the contents of the database table into MATLAB again and display the last few rows. The results contain a new row for the inserted sales volume data.

```
rows = sqlread(conn, tablename);
tail(rows, 4)
```

ans =

4×13 table

StockNumber	January	February	March	April	May	June	July	August	Sep
4.7082e+05	3100	9400	1540	1500	1350	1190	900	867	
5.101e+05	235	1800	1040	900	750	700	400	350	
8.9975e+05	123	1700	823	701	689	621	545	421	
1e+05	NaN	0	2000	500	3000	450	600	700	

Close the database connection.

```
close(conn)
```

Insert NULL String into Table

Use an ODBC connection to insert product data from MATLAB® into an existing table in a Microsoft® SQL Server® database. Insert a NULL string into the existing database table.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the table `productTable`.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Create a MATLAB table that contains data for one product and includes a NULL value in the `productDescription` variable.

```
data = table([30],[500000],[1000],[25], ...
    ["null"],'VariableNames',{'productNumber' ...
    'stockNumber' 'supplierNumber' 'unitCost' 'productDescription'});
```

Convert the null value in the `productDescription` variable to "". The `sqlwrite` function does not accept null values as valid missing data for insertion.

```
data.productDescription(1) = "";
```

Import the contents of the existing database table `productTable` into MATLAB and display the last few rows.

```
tablename = 'productTable';
rows = sqlread(conn,tablename);
tail(rows,3)
```

```
ans =
```

```
3×5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	4.7082e+05	1012	17	'Pancakes'
14	5.101e+05	1011	19	'Shawl'
15	8.9975e+05	1011	20	'Snacks'

Insert the product data into the database table `productTable`.

```
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB again and display the last few rows. The results contain a new row for the inserted product.

```
rows = sqlread(conn,tablename);
tail(rows,4)
```

```
ans =
```

```
4x5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	4.7082e+05	1012	17	'Pancakes'
14	5.101e+05	1011	19	'Shawl'
15	8.9975e+05	1011	20	'Snacks'
30	5e+05	1000	25	''

Close the database connection.

```
close(conn)
```

Insert Partial Data into Table

Use an ODBC connection to insert two columns of inventory data from MATLAB® into an existing table with a few columns in a Microsoft® SQL Server® database. After the insertion of data, the database table contains the data in the first two columns. The other columns have blank entries for the inserted rows of data.

Create an ODBC connection to a SQL Server database with Windows® authentication. Specify a blank user name and password. The database contains the table `inventoryTable` with inventory data.

```
datasource = "MS SQL Server Auth";
conn = database(datasource, "", "");
```

Create a table with two columns of data for the product number and quantity. This data represents two products to insert.

```
data = table([14;15],[350;400], 'VariableNames', ["productNumber" "Quantity"]);
```

Insert the inventory data into the database table `inventoryTable`. This table contains four columns: product number, quantity, price, and inventory date. In this case, the product number and quantity for each product to insert are known but the price and inventory date are unknown.

```
tablename = "inventoryTable";
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the last few rows. The results contain two new rows for the inserted inventory data. The `sqlwrite` function inserts blank entries for the last two columns.

```
rows = sqlread(conn,tablename);
tail(rows,3)
```

```
ans=3x4 table
   productNumber   Quantity   Price   inventoryDate
   _____   _____   _____   _____
           13           1700      14.5   {'2009-05-24 10:58:59'}
           14            350       NaN   {0x0 char           }
           15            400       NaN   {0x0 char           }
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

data — Data to insert

table

Data to insert into a database table, specified as a table.

The valid data types in a MATLAB table are:

- Numeric array
- Cell array of numeric arrays
- Cell array of character vectors
- String array
- Datetime array
- Duration array
- Logical array

- Cell array of logical arrays

The numeric array can contain these data types:

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `int64`
- `uint64`
- `single`
- `double`

For date and time data, supported formats are:

- Date — `'yyyy-MM-dd'`
- Time — `'hh:mm:ss'`
- Timestamp — `'yyyy-MM-dd HH:mm:ss'`

If the date and time data is specified in an invalid format, then the `sqlwrite` function automatically converts the data to a supported format.

If the cell array of character vectors or string array is specified in an invalid format, then the `sqlwrite` function enables the database driver to check the format. If the format is unexpected, then the database driver throws an error.

You can insert data in an existing database table or a new database table. The data types of variables in `data` vary depending on whether the database table exists. For valid data types, see “Data Types for Existing Table” on page 11-0 and “Data Types for New Table” on page 11-0 .

Note The `sqlwrite` function supports only the `table` data type for the `data` input argument. To insert data stored in a structure, cell array, or numeric matrix, convert the data to a `table` by using the `struct2table`, `cell2table`, and `array2table` functions, respectively.

The `sqlwrite` function does not support the database preferences `NullNumberWrite` and `NullStringWrite`. To insert missing data, see “Accepted Missing Data” on page 11-0 .

Example: `table([10;20],{'M';'F'})`

Data Types for Existing Table

The variable names of the MATLAB table must match the column names in the database table. The `sqlwrite` function is case-sensitive.

When you insert data into a database table, use the data types shown in the following table to ensure that the data has the correct data type. This table matches the valid data types of the MATLAB table variable to the data types of the database column. For example, when you insert data into a database

column that has the `BOOLEAN` data type, ensure that the corresponding variable in the MATLAB table is a logical array or cell array of logical arrays.

Data Type of MATLAB Table Variable	Data Type of Existing Database Column
Numeric array or cell array of numeric arrays	NUMERIC
<ul style="list-style-type: none"> Cell array of character vectors String array Datetime array Duration array 	DATE, TIME, or DATETIME
Logical array or cell array of logical arrays	BIT or BOOLEAN
Cell array of character vectors or string array	<ul style="list-style-type: none"> CHAR VARCHAR TEXT NTEXT Other text data type

Data Types for New Table

The specified table name for the new database table must be unique across all tables in the database.

The valid data types in a MATLAB table are:

- Numeric array
- Cell array of character vectors
- String array
- Datetime array
- Duration array
- Logical array

The `sqlwrite` function ignores any invalid variable types and inserts only the valid variables from MATLAB as columns in a new database table.

The `sqlwrite` function converts the data type of the variable into the default data type of the column in the database table. The following table matches the valid data types of the MATLAB table variable to the default data types of the database column.

Data Type of MATLAB Table Variable	Default Data Type of Database Column
Numeric array or cell array of numeric arrays	NUMERIC
Datetime array	TIMESTAMP
Duration array	TIME
Logical array	NUMERIC

Data Type of MATLAB Table Variable	Default Data Type of Database Column
String array	VARCHAR Note The size of this column equals the sum of the maximum length of a string in the string array and 100.
Cell array of character vectors	VARCHAR Note The size of this column equals the sum of the maximum length of a character vector in the cell array and 100.

To specify database-specific column data types instead of the defaults, use the 'ColumnType' name-value pair argument. For example, you can specify 'ColumnType', "bigint" to create a BIGINT column in the new database table.

Also, using the 'ColumnType' name-value pair argument, you can specify other data types that are not in the default list. For example, to insert images, specify 'ColumnType', "image".

Accepted Missing Data

The accepted missing data for inserting data into a database depends on the data type of the MATLAB table variable and the data type of the column in the database. The following table matches the data type of the MATLAB table variable to the data type of the database column and specifies the accepted missing data to use in each case.

Data Type of MATLAB Table Variable	Data Type of Database Column	Accepted Missing Data
datetime array	Date, Time, or Timestamp	NaT
duration array	Time	NaN
double or single array	<ul style="list-style-type: none"> • Numeric • Double • Float • Decimal • Real 	NaN
cell array of double or single arrays	<ul style="list-style-type: none"> • Numeric • Double • Float • Decimal • Real 	NaN, [], or ''
cell array of character vectors	Date, Time, or Timestamp	'NaT' or ''
cell array of character vectors	Char, Varchar, or other text data type	''
string array	Date, Time, or Timestamp	"" , "NaT", or missing

Data Type of MATLAB Table Variable	Data Type of Database Column	Accepted Missing Data
string array	Char, Varchar, or other text data type	missing

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `sqlwrite(conn,"tablename",data,'ColumnType',['numeric' "timestamp" "image"])` inserts data into a new database table named `tablename` by specifying data types for all columns in the new database table.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: `string` | `char`

Schema — Database schema name

string scalar | character vector

Database schema name, specified as a string scalar or character vector. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Example: `Schema = "dbo"`

Data Types: `string` | `char`

ColumnType — Database column types

character vector | string scalar | cell array of character vectors | string array

Database column types, specified as a character vector, string scalar, cell array of character vectors, or string array. Use this name-value pair argument to define custom data types for the columns in a database table. Specify a column type for each column in the table.

Example: `'ColumnType',["numeric" "varchar(400)"]`

Data Types: `char` | `string` | `cell`

Version History

Introduced in R2018a

See Also

sqlread | database | close | cell2table | array2table | struct2table

Topics

“Insert Data into Database Table” on page 5-61

“Append Data to Existing Database Table Using Insert Functionality” on page 5-53

“Insert Data into New Database Table Using Insert Functionality” on page 5-55

“Writing Data Common Errors” on page 3-2

sqlwrite

Insert MATLAB data into SQLite database table

Syntax

```
sqlwrite(conn,tablename,data)
sqlwrite(conn,tablename,data,ColumnType=columntypes)
```

Description

`sqlwrite(conn,tablename,data)` inserts data from a MATLAB table into a database table with the MATLAB interface to SQLite. If the table exists in the database, this function appends the data in the MATLAB table as rows in the existing database table. If the table does not exist in the database, this function creates a table with the specified table name and then inserts the data as rows in the new table. This syntax is the equivalent of executing SQL statements that contain the `CREATE TABLE` and `INSERT INTO` ANSI SQL syntaxes.

`sqlwrite(conn,tablename,data,ColumnType=columntypes)` specifies the data type for the column in the SQLite database.

Examples

Append Data into Existing Table

Use the MATLAB® interface to SQLite to append product data from a MATLAB® table into an existing table in an SQLite database.

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

To view the existing database table `productTable` before appending data, import its contents into MATLAB and display the last few rows.

```
tablename = "productTable";
rows = sqlread(conn,tablename);
tail(rows,3)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	470816	1012	16	"Pancakes"
14	510099	1011	19	"Shawl"
15	899752	1011	20	"Snacks"

Create a MATLAB table that contains the data for one product.

```
data = table(30,500000,1000,25,"Rubik's Cube", ...
    'VariableNames',["productNumber" "stockNumber" ...
    "supplierNumber" "unitCost" "productDescription"]);
```

Append the product data into the database table `productTable`.

```
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB again and display the last few rows. The results contain a new row for the inserted product.

```
rows = sqlread(conn,tablename);
tail(rows,4)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
13	470816	1012	16	"Pancakes"
14	510099	1011	19	"Shawl"
15	899752	1011	20	"Snacks"
30	500000	1000	25	"Rubik's Cube"

Delete the inserted row to maintain the dataset.

```
sqlquery = "DELETE FROM productTable WHERE productnumber = 30";
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Insert Data into New Table

Use the MATLAB® interface to SQLite to insert product data from MATLAB into a new table in an SQLite database.

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Create a MATLAB table that contains data for two products.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"],'VariableNames',["productNumber" ...
    "stockNumber" "supplierNumber" "unitCost" "productDescription"]);
```

Insert the product data into a new database table named `toyTable`.

```
tablename = "toyTable";
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
  _____  _____  _____  _____  _____
           30           5e+05           1000           25           "Rubik's Cube"
           40           6e+05           2000           30           "Doll House"
```

Delete the new table to maintain the dataset.

```
sqlquery = "DROP TABLE toyTable";
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Specify Column Types When Inserting Data into New Table

Use the MATLAB® interface to SQLite to insert product data from MATLAB into a new table in an SQLite database. Specify the data types of the columns in the new database table.

Create the SQLite connection `conn` to the existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. The SQLite connection is an `sqlite` object.

```
dbfile = "tutorial.db";
conn = sqlite(dbfile);
```

Create a MATLAB table that contains data for two products.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
  ["Rubik's Cube";"Doll House"],'VariableNames',["productNumber" ...
  "stockNumber" "supplierNumber" "unitCost" "productDescription"]);
```

Insert the product data into a new database table named `toyTable`. Use the `ColumnType` name-value argument and a string array to specify the data types of all the columns in the database table.

```
tablename = "toyTable";
coltypes = ["numeric" "numeric" "numeric" "numeric" "varchar(255)"];
sqlwrite(conn,tablename,data,ColumnType=coltypes)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
  _____  _____  _____  _____  _____
           30           500000           1000           25           "Rubik's Cube"
           40           600000           2000           30           "Doll House"
```

Delete the new table to maintain the dataset.

```
sqlquery = "DROP TABLE toyTable";  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — SQLite database connection

sqlite object

SQLite database connection, specified as an `sqlite` object created using the `sqlite` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: `string` | `char`

data — Data to insert

table

Data to insert into a database table, specified as a table.

The valid data types in a MATLAB table are:

- Numeric array
- Cell array of numeric arrays
- Cell array of character vectors
- String array
- Datetime array
- Logical array
- Cell array of logical arrays

The numeric array can contain these data types:

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `int64`

- `uint64`
- `single`
- `double`

For date and time data, supported formats are:

- Date — `'yyyy-MM-dd'`
- Time — `'hh:mm:ss'`
- Timestamp — `'yyyy-MM-dd HH:mm:ss'`

If the date and time data is specified in an invalid format, then the `sqlwrite` function automatically converts the data to a supported format.

If the cell array of character vectors or string array is specified in an invalid format, then the `sqlwrite` function enables the database driver to check the format. If the format is unexpected, then the database driver throws an error.

You can insert data in an existing database table or a new database table. The data types of variables in `data` vary depending on whether the database table exists. For valid data types, see “Data Types for Existing Table” on page 11-0 and “Data Types for New Table” on page 11-0 .

Note The `sqlwrite` function supports only the `table` data type for the data input argument. To insert data stored in a structure, cell array, or numeric matrix, convert the data to a `table` by using the `struct2table`, `cell2table`, and `array2table` functions, respectively.

Example: `table([10;20],{'M';'F'})`

Data Types for Existing Table

The variable names of the MATLAB table must match the column names in the database table. The `sqlwrite` function is case-sensitive.

When you insert data into a database table, use the data types shown in the following table to ensure that the data has the correct data type. This table matches the valid data types of the MATLAB table variable to the data types of the database column. For example, when you insert data into a database column that has the `BOOLEAN` data type, ensure that the corresponding variable in the MATLAB table is a logical array or cell array of logical arrays.

Data Type of MATLAB Table Variable	Data Type of Existing Database Column
Numeric array or cell array of numeric arrays	DOUBLE
<ul style="list-style-type: none"> • String array • Cell array of character vectors • Datetime array 	VARCHAR
Logical array	DOUBLE

Data Types for New Table

The specified table name for the new database table must be unique across all tables in the database.

The valid data types in a MATLAB table are:

- Numeric array
- Cell array of character vectors
- String array
- Datetime array
- Logical array

The `sqlwrite` function ignores any invalid variable types and inserts only the valid variables from MATLAB as columns in a new database table.

The `sqlwrite` function converts the data type of the variable into the default data type of the column in the database table. The following table matches the valid data types of the MATLAB table variable to the default data types of the database column.

Data Type of MATLAB Table Variable	Default Data Type of Database Column
Numeric array or cell array of numeric arrays	DOUBLE
<ul style="list-style-type: none"> • String array • Cell array of character vectors • Datetime array 	VARCHAR
Logical array	DOUBLE

To specify database-specific column data types instead of the defaults, use the `ColumnType` name-value argument. For example, you can specify `ColumnType="DOUBLE"` to create a `DOUBLE` column in the new database table.

Accepted Missing Data

The accepted missing data for inserting data into a database depends on the data type of the MATLAB table variable and the data type of the column in the database. The following table matches the data type of the MATLAB table variable to the data type of the database column and specifies the accepted missing data to use in each case.

Data Type of MATLAB Table Variable	Accepted Missing Data
Numeric array	NaN
String array	missing
Cell array of character vectors	' '
Datetime array	NaT

Data Types: table

columntypes — Database column types

character vector | string scalar | cell array of character vectors | string array

Database column types, specified as a character vector, string scalar, cell array of character vectors, or string array. Use this argument to define custom data types for the columns in a database table. Specify a column type for each column in the table.

Example: `["numeric" "varchar(400)"]`

Data Types: cell | char | string

Version History

Introduced in R2022a

See Also

Objects

sqlite

Functions

sqlread | close

Topics

“Insert Data into SQLite Database Table” on page 5-36

“Create Table and Add Column in SQLite Database” on page 5-38

“Delete Data from SQLite Database” on page 5-39

“Roll Back Data in SQLite Database” on page 5-41

SQLImportOptions

Define import options for database data

Description

After you create an `SQLImportOptions` object, you can customize the import options for importing data from a database into MATLAB. Import options include defining the data types and fill values for missing data.

Creation

Create an `SQLImportOptions` object with the `databaseImportOptions` function.

Properties

ExcludeDuplicates — Flag to exclude duplicates

`false` (default) | `true`

Flag to exclude duplicates from imported data, specified as `false` or `true`. To exclude duplicates from the data in a database table or the results of an SQL query, set the `ExcludeDuplicates` property to `true` using dot notation.

Setting this property is the equivalent of using the `DISTINCT` SQL statement in ANSI SQL.

Data Types: `logical`

VariableNames — Variable names

cell array of character vectors

Variable names, specified as a cell array of character vectors. Each character vector in the cell array indicates the name of an imported database column from an SQL query or database table.

For a table or SQL query with only one database column, the cell array contains only one character vector.

The default variable names are the names of the columns in an SQL query or database table.

Example: `{ 'productNumber', 'stockNumber' }`

Data Types: `cell`

VariableTypes — Variable types

cell array of character vectors

Variable types, specified as a cell array of character vectors. Each character vector in the cell array indicates the data type of an imported database column from an SQL query or database table. Each character vector must be a valid MATLAB data type.

For a table or SQL query with only one database column, the cell array contains only one character vector.

When you create the `SQLImportOptions` object, the `databaseImportOptions` function automatically detects the data type based on the data type of a database column. This table maps the data type of a database column to the detected MATLAB data type.

Database Data Type	MATLAB Detected Data Type
TEXT	char
DATE, TIME, DATETIME, or TIMESTAMP	char
NUMERIC	double
BOOLEAN or BIT	logical

If you are using the MySQL native interface, this table maps the data type of a database column to the detected MATLAB data type.

MySQL Data Type	MATLAB Data Type
BIT	logical
TINYINT	double
SMALLINT	double
BIGINT	double
REAL	double
DOUBLE	double
DECIMAL	double
NUMERIC	double
CHAR	string
VARCHAR	string
LONGVARCHAR	string
TIMESTAMP	datetime
DATE	datetime
TIME	duration
YEAR	double
ENUM	categorical
JSON	char

If you are using the PostgreSQL native interface, this table maps the data type of a database column to the detected MATLAB data type.

PostgreSQL Data Type	MATLAB Data Type
Boolean	logical
Smallint	double
Integer	double
Bigint	double
Decimal	double

PostgreSQL Data Type	MATLAB Data Type
Numeric	double
Real	double
Double Precision	double
Smallserial	double
Serial	double
Bigserial	double
Money	double
Varchar	string
Char	string
Text	string
Bytea	string
Timestamp	datetime
Timestampz	datetime
Abstime	datetime
Date	datetime
Time	duration
Timez	duration
Interval	calendarDuration
Reltime	calendarDuration
Enum	categorical
Cidr	string
Inet	string
Macaddr	string
Uuid	string
Xml	string

To update the `VariableTypes` property, use the `setoptions` function.

Example: `{'int64','int32'}`

Data Types: `cell`

SelectedVariableNames — Subset of variables to import

character vector | cell array of character vectors | numeric array

Subset of variables to import, specified as a character vector, cell array of character vectors, or numeric array that contains indices. Use the `SelectedVariableNames` property to determine the database columns to import into the MATLAB workspace.

The values in the `SelectedVariableNames` property must be equal to the values in the `VariableNames` property or a subset of these values. By default, the `SelectedVariableNames` property contains all variable names specified in the `VariableNames` property. When the

`SelectedVariableNames` property specifies all variable names, the `sqlread`, `fetch`, and `import` functions of the `DatabaseDatastore` object import all database columns.

Example: `{'productNumber', 'stockNumber'}`

Example: `[1,2,3]`

Data Types: `double` | `char` | `cell`

FillValues – Fill value for missing data

cell array

Fill value for missing data, specified as a cell array that contains one or more values. Each value can be one of these data types:

- All integer classes
- `single`
- `double`
- `char`
- `string scalar`
- `logical`
- `datetime array`
- `categorical array`
- `missing`

When you create the `SQLImportOptions` object, the `databaseImportOptions` function automatically detects the fill value for missing data based on the data type of the database column. This table maps the data type of a database column to the detected MATLAB fill value.

Database Data Type	MATLAB Detected Fill Value
TEXT	' '
DATE, TIME, DATETIME, or TIMESTAMP	' '
NUMERIC	NaN
BOOLEAN or BIT	false

If you are using the MySQL native interface, this table maps the data type of a database column to the detected MATLAB data type.

MySQL Data Type	MATLAB Detected Fill Value
CHAR, VARCHAR, LONGVARCHAR, or JSON	' ' (if the <code>VariableTypes</code> property is <code>char</code>) or <code><missing></code> (if the <code>VariableTypes</code> property is <code>string</code>)
DATE or TIMESTAMP	NaN
TIME	NaN

MySQL Data Type	MATLAB Detected Fill Value
<ul style="list-style-type: none"> • TINYINT • SMALLINT • BIGINT • REAL • DOUBLE • DECIMAL • NUMERIC • YEAR 	NaN
ENUM	<undefined>

If you are using the PostgreSQL native interface, this table maps the data type of a database column to the detected MATLAB data type.

PostgreSQL Data Type	MATLAB Detected Fill Value
boolean	false
<ul style="list-style-type: none"> • smallint • integer • bigint • decimal • numeric • real • double precision 	NaN
char, varchar, or text	<missing>
date or timestamp	NaT
time	NaN
interval	NaN
enum	undefined

To update the FillValues property, use the `setoptions` function.

Example: `{ ' ', NaN }`

Data Types: `cell`

VariableOptions — Type-specific variable import options

array of variable import options objects

Type-specific variable import options, returned as an array of variable import options objects. The array contains an object corresponding to each variable specified in the `VariableNames` property. Each object in the array contains properties that support the importing of data with a specific data type.

To query the current (or detected) options for a variable, use the `getoptions` function.

To set and customize options for a variable, use the `setoptions` function.

Example: `opts.VariableOptions` returns a collection of `SQLVariableImportOptions` objects, one corresponding to each variable in the data.

VariableNamingRule — Variable naming rule

"modify" (default) | "preserve"

Variable naming rule, specified as one of these values:

- "modify" — Remove non-ASCII characters from variable names when the `SQLImportOptions` function imports data.
- "preserve" — Preserve most variable names when the `SQLImportOptions` function imports data. For details, see "Limitations" on page 11-444.

If you are using the MySQL or PostgreSQL native interface, "preserve" is the default value.

The `VariableNamingRule` property has these limitations:

- The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the `table` data type.
- The length of each variable name must be less than the number returned by `namelengthmax`.

Example: "VariableNamingRule", "modify"

Data Types: string

RowFilter — Row filter condition

<unconstrained> (default) | `matlab.io.RowFilter` object

Row filter condition, specified as a `matlab.io.RowFilter` object. Using `RowFilter` you can filter by columns or other criteria not listed in the `VariableNames` property. To do this, set a new `RowFilter` property containing the new variable name. See "Filter Data by Using Unique Variable Names" on page 11-442.

Example: `opt.RowFilter = opt.RowFilter.productnumber <= 5`

Object Functions

<code>getoptions</code>	Retrieve import options for database data
<code>preview</code>	Preview eight rows from database using import options
<code>reset</code>	Reset to default import options for database data
<code>setoptions</code>	Customize import options for database data

Examples

Import Data from Database Table Using Import Options

Customize import options when importing data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize import options for different database columns. Import data using the `sqlread` function.

This example uses the `patients.xls` file, which contains the columns `Gender`, `Location`, `SelfAssessedHealthStatus`, and `Smoker`. The example also uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank username and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the patients database table using the patient information.

```
tablename = 'patients';
sqlwrite(conn, tablename, patients)
```

Create an SQLImportOptions object using the patients database table and the databaseImportOptions function.

```
opts = databaseImportOptions(conn, tablename)
```

```
opts =
  SQLImportOptions with properties:

    ExcludeDuplicates: false
    VariableNamingRule: 'modify'

    VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    VariableTypes: {'char', 'char', 'double' ... and 7 more}
    SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    FillValues: {'', '', NaN ... and 7 more }
    RowFilter: <unconstrained>

    VariableOptions: Show all 10 VariableOptions
```

Display the current import options for the variables selected in the SelectedVariableNames property of the SQLImportOptions object.

```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts, vars)
```

```
varOpts =
  1x10 SQLVariableImportOptions array with properties:

    Variable Options:
           (1) |      (2) |      (3) |      (4) |      (5) |      (6) |      (7) |
    Name: 'LastName' | 'Gender' | 'Age' | 'Location' | 'Height' | 'Weight' | 'Smoker' |
    Type: 'char' | 'char' | 'double' | 'char' | 'double' | 'double' | 'double' |
    MissingRule: 'fill' | 'fill' | 'fill' | 'fill' | 'fill' | 'fill' | 'fill' |
    FillValue: '' | '' | NaN | '' | NaN | NaN | NaN |
```

To access sub-properties of each variable, use getoptions

Change the data types for the Gender, Location, SelfAssessedHealthStatus, and Smoker variables using the setoptions function. Because the Gender, Location, and SelfAssessedHealthStatus variables indicate a finite set of repeating values, change their data type to categorical. Because the Smoker variable stores the values 0 and 1, change its data type to logical. Then, display the updated import options.

```
opts = setoptions(opts,{'Gender','Location','SelfAssessedHealthStatus'}, ...
    'Type','categorical');
opts = setoptions(opts,'Smoker','Type','logical');
```

```
varOpts = getoptions(opts,{'Gender','Location','Smoker', ...
    'SelfAssessedHealthStatus'})
```

```
varOpts =
    1x4 SQLVariableImportOptions array with properties:
```

Variable Options:

	(1)	(2)	(3)	(4)
Name:	'Gender'	'Location'	'Smoker'	'SelfAssessedHealthStatus'
Type:	'categorical'	'categorical'	'logical'	'categorical'
MissingRule:	'fill'	'fill'	'fill'	'fill'
FillValue:	<undefined>	<undefined>	0	<undefined>

To access sub-properties of each variable, use `getoptions`

Import the `patients` database table using the `sqlread` function, and display the last eight rows of the table.

```
data = sqlread(conn,tablename,opts);
tail(data)
```

LastName	Gender	Age	Location	Height	Weight	Smoker
{'Foster' }	Female	30	St. Mary's Medical Center	70	124	false
{'Gonzales' }	Male	48	County General Hospital	71	174	false
{'Bryant' }	Female	48	County General Hospital	66	134	false
{'Alexander' }	Male	25	County General Hospital	69	171	true
{'Russell' }	Male	44	VA Hospital	69	188	true
{'Griffin' }	Male	49	County General Hospital	70	186	false
{'Diaz' }	Male	45	County General Hospital	68	172	true
{'Hayes' }	Male	48	County General Hospital	66	177	false

Display a summary of the imported data. The `sqlread` function applies the import options to the variables in the imported data.

```
summary(data)
```

Variables:

LastName: 100×1 cell array of character vectors

Gender: 100×1 categorical

Values:

Female	53
Male	47

Age: 100×1 double

Values:

Min	25
Median	39

Max 50

Location: 100×1 categorical

Values:

County General Hospital	39
St. Mary s Medical Center	24
VA Hospital	37

Height: 100×1 double

Values:

Min	60
Median	67
Max	72

Weight: 100×1 double

Values:

Min	111
Median	142.5
Max	202

Smoker: 100×1 logical

Values:

True	34
False	66

Systolic: 100×1 double

Values:

Min	109
Median	122
Max	138

Diastolic: 100×1 double

Values:

Min	68
Median	81.5
Max	99

SelfAssessedHealthStatus: 100×1 categorical

Values:

Excellent	34
Fair	15
Good	40
Poor	11

Now set the filter condition to import only data for patients older than 40 years and not taller than 68 inches.

```
opts.RowFilter = opts.RowFilter.Age > 40 & opts.RowFilter.Height <= 68
```

```
opts =
  SQLImportOptions with properties:
    ExcludeDuplicates: false
    VariableNamingRule: 'modify'
    VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    VariableTypes: {'char', 'categorical', 'double' ... and 7 more}
    SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    FillValues: {'', <undefined>, NaN ... and 7 more }
    RowFilter: Age > 40 & Height <= 68
    VariableOptions: Show all 10 VariableOptions
```

Again, import the patients database table using the `sqlread` function, and display a summary of the imported data.

```
data = sqlread(conn,tablename,opts);
summary(data)
```

Variables:

LastName: 24×1 cell array of character vectors

Gender: 24×1 categorical

Values:

Female	17
Male	7

Age: 24×1 double

Values:

Min	41
Median	45.5
Max	50

Location: 24×1 categorical

Values:

County General Hospital	13
St. Mary s Medical Center	5
VA Hospital	6

Height: 24×1 double

Values:

Min	62
-----	----

Median	66
Max	68

Weight: 24×1 double

Values:

Min	119
Median	137
Max	194

Smoker: 24×1 logical

Values:

True	8
False	16

Systolic: 24×1 double

Values:

Min	114
Median	121.5
Max	138

Diastolic: 24×1 double

Values:

Min	68
Median	81.5
Max	96

SelfAssessedHealthStatus: 24×1 categorical

Values:

Excellent	7
Fair	3
Good	10
Poor	4

Delete the patients database table using the execute function.

```
sqlquery = ['DROP TABLE ' tablename];  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Filter Data by Using Unique Variable Names

Filter data by columns or other criteria not listed in the VariableNames property by using the RowFilter property.

Create a PostgreSQL native interface database connection to a PostgreSQL database. The database contains the table `productTable`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";

conn = postgresql(datasource,username,password);
```

Import data from the database table `productTable`. The `sqlread` function returns a MATLAB table that contains the product data. Display the first five rows of product data.

```
tablename = "productTable";
data = sqlread(conn,tablename);
head(data,5)
```

productnumber	stocknumber	suppliernumber	unitcost	productdescription
9	1.2597e+05	1003	13	"Victorian Doll"
8	2.1257e+05	1001	5	"Train Set"
7	3.8912e+05	1007	16	"Engine Kit"
2	4.0031e+05	1002	9	"Painting Set"
4	4.0034e+05	1008	21	"Space Cruiser"

Create an `SQLImportOptions` object using the `productTable` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename)
```

```
opts =
  SQLImportOptions with properties:
    ExcludeDuplicates: false
    VariableNamingRule: 'preserve'
    VariableNames: {'productnumber', 'stocknumber', 'suppliernumber' ... and 2 more}
    VariableTypes: {'double', 'double', 'double' ... and 2 more}
    SelectedVariableNames: {'productnumber', 'stocknumber', 'suppliernumber' ... and 2 more}
    FillValues: { NaN, NaN, NaN ... and 2 more }
    RowFilter: <unconstrained>

    VariableOptions: Show all 5 VariableOptions
```

Set the `RowFilter` property containing the new variable name. For example, use the `length` function to filter the `productdescription` strings by the number of characters.

```
opts.RowFilter = rowfilter("length(productdescription)");
opts.RowFilter
```

```
ans =
  RowFilter with no constraints and no selected variables
  <unconstrained>
  VariableNames: length(productdescription), productnumber, stocknumber, suppliernumber, unitcost
```

By setting `RowFilter`, you added a unique `VariableName` to `RowFilter`. The `VariableNames` property of `SQLImportOptions` does not contain this variable name. Even though `VariableNames` of `RowFilter` does not completely match `VariableNames` of `SQLImportOptions`, updating `VariableNames` of `SQLImportOptions` still updates `VariableNames` of `RowFilter`.

```
opts.VariableNames{4} = "cost";
opts.RowFilter
```

```
ans =
```

```
RowFilter with no constraints and no selected variables
```

```
<unconstrained>
```

```
VariableNames: length(productdescription), productnumber, stocknumber, suppliernumber, cost, p
```

Set the filtering condition using the unique variable name, `length(productdescription)`, and the new variable name, `cost`.

```
opts.RowFilter = opts.RowFilter.("length(productdescription)") < 10 & opts.RowFilter.cost > 10;
```

Import data from the database table and display the first five rows of product data.

```
data = sqlread(conn,tablename,opts);
head(data,5)
```

productnumber	stocknumber	suppliernumber	cost	productdescription
3	4.01e+05	1009	17	"Slinky"
12	2.1046e+05	1010	22	"Hugsy"
13	4.7082e+05	1012	16.5	"Pancakes"
14	5.101e+05	1011	19	"Shawl"
15	8.9975e+05	1011	20	"Snacks"

Limitations

- If you use the "VariableNamingRule" name-value argument with the `SQLImportOptions` object `opts`, the data import functions return an error.
- If you set the `VariableNamingRule` name-value argument to the value "modify":
 - The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the `table` data type.
 - The length of each variable name must be less than the number returned by `namelengthmax`.
- The `fetch` and `sqlread` functions return an error if you specify the `RowFilter` name-value argument with the `SQLImportOptions` object `opts`. In this case, it is ambiguous which of the `RowFilter` object to use, especially if the filter conditions are different.

Version History

Introduced in R2018b

R2023a: Selectively import rows of data

Use the `RowFilter` property of `SQLImportOptions` to define import options for database data.

See Also

Functions

`sqlwrite` | `database` | `close` | `sqlread` | `fetch` | `execute` | `databaseDatastore`

Topics

“Customize Options for Importing Data from Database into MATLAB” on page 5-67

“Importing Data Common Errors” on page 3-3

External Websites

SQL Tutorial

databaseImportOptions

Define import options for database data

Syntax

```
opts = databaseImportOptions(conn,source)
opts = databaseImportOptions(conn,source,Name,Value)
```

Description

`opts = databaseImportOptions(conn,source)` creates an `SQLImportOptions` object using the database connection and a source, which is a database table name or SQL query.

`opts = databaseImportOptions(conn,source,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'Catalog','toystore_doc'` retrieves data from the `toystore_doc` database catalog.

Examples

Import Data from Database Table Using Import Options

Customize import options when importing data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize import options for different database columns. Import data using the `sqlread` function.

This example uses the `patients.xls` file, which contains the columns `Gender`, `Location`, `SelfAssessedHealthStatus`, and `Smoker`. The example also uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank username and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';
sqlwrite(conn,tablename,patients)
```

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename)
```

```
opts =
    SQLImportOptions with properties:
```

```

ExcludeDuplicates: false
VariableNamingRule: 'modify'

VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
VariableTypes: {'char', 'char', 'double' ... and 7 more}
SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
FillValues: {'', '', NaN ... and 7 more }
RowFilter: <unconstrained>

VariableOptions: Show all 10 VariableOptions

```

Display the current import options for the variables selected in the `SelectedVariableNames` property of the `SQLImportOptions` object.

```

vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)

```

```

varOpts =
    1x10 SQLVariableImportOptions array with properties:

```

```

Variable Options:

```

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Name:	'LastName'	'Gender'	'Age'	'Location'	'Height'	'Weight'	'Smoker'
Type:	'char'	'char'	'double'	'char'	'double'	'double'	'double'
MissingRule:	'fill'	'fill'	'fill'	'fill'	'fill'	'fill'	'fill'
FillValue:	''	''	NaN	''	NaN	NaN	NaN

To access sub-properties of each variable, use `getoptions`

Change the data types for the `Gender`, `Location`, `SelfAssessedHealthStatus`, and `Smoker` variables using the `setoptions` function. Because the `Gender`, `Location`, and `SelfAssessedHealthStatus` variables indicate a finite set of repeating values, change their data type to `categorical`. Because the `Smoker` variable stores the values 0 and 1, change its data type to `logical`. Then, display the updated import options.

```

opts = setoptions(opts,{'Gender','Location','SelfAssessedHealthStatus'}, ...
    'Type','categorical');
opts = setoptions(opts,'Smoker','Type','logical');

```

```

varOpts = getoptions(opts,{'Gender','Location','Smoker', ...
    'SelfAssessedHealthStatus'})

```

```

varOpts =
    1x4 SQLVariableImportOptions array with properties:

```

```

Variable Options:

```

	(1)	(2)	(3)	(4)
Name:	'Gender'	'Location'	'Smoker'	'SelfAssessedHealthStatus'
Type:	'categorical'	'categorical'	'logical'	'categorical'
MissingRule:	'fill'	'fill'	'fill'	'fill'
FillValue:	<undefined>	<undefined>	0	<undefined>

To access sub-properties of each variable, use `getoptions`

Import the `patients` database table using the `sqlread` function, and display the last eight rows of the table.

```
data = sqlread(conn,tablename,opts);
tail(data)
```

LastName	Gender	Age	Location	Height	Weight	Smoker
{'Foster' }	Female	30	St. Mary's Medical Center	70	124	false
{'Gonzales' }	Male	48	County General Hospital	71	174	false
{'Bryant' }	Female	48	County General Hospital	66	134	false
{'Alexander' }	Male	25	County General Hospital	69	171	true
{'Russell' }	Male	44	VA Hospital	69	188	true
{'Griffin' }	Male	49	County General Hospital	70	186	false
{'Diaz' }	Male	45	County General Hospital	68	172	true
{'Hayes' }	Male	48	County General Hospital	66	177	false

Display a summary of the imported data. The `sql read` function applies the import options to the variables in the imported data.

```
summary(data)
```

Variables:

LastName: 100×1 cell array of character vectors

Gender: 100×1 categorical

Values:

Female	53
Male	47

Age: 100×1 double

Values:

Min	25
Median	39
Max	50

Location: 100×1 categorical

Values:

County General Hospital	39
St. Mary s Medical Center	24
VA Hospital	37

Height: 100×1 double

Values:

Min	60
Median	67
Max	72

Weight: 100×1 double

Values:

Min	111
Median	142.5
Max	202

Smoker: 100×1 logical

Values:

True	34
False	66

Systolic: 100×1 double

Values:

Min	109
Median	122
Max	138

Diastolic: 100×1 double

Values:

Min	68
Median	81.5
Max	99

SelfAssessedHealthStatus: 100×1 categorical

Values:

Excellent	34
Fair	15
Good	40
Poor	11

Now set the filter condition to import only data for patients older than 40 years and not taller than 68 inches.

```
opts.RowFilter = opts.RowFilter.Age > 40 & opts.RowFilter.Height <= 68
```

```
opts =
```

```
  SQLImportOptions with properties:
```

```
    ExcludeDuplicates: false
    VariableNamingRule: 'modify'
```

```
      VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
      VariableTypes: {'char', 'categorical', 'double' ... and 7 more}
    SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
      FillValues: {'', <undefined>, NaN ... and 7 more }
      RowFilter: Age > 40 & Height <= 68
```

```
    VariableOptions: Show all 10 VariableOptions
```

Again, import the patients database table using the `sqlread` function, and display a summary of the imported data.

```
data = sqlread(conn,tablename,opts);  
summary(data)
```

Variables:

LastName: 24×1 cell array of character vectors

Gender: 24×1 categorical

Values:

Female	17
Male	7

Age: 24×1 double

Values:

Min	41
Median	45.5
Max	50

Location: 24×1 categorical

Values:

County General Hospital	13
St. Mary s Medical Center	5
VA Hospital	6

Height: 24×1 double

Values:

Min	62
Median	66
Max	68

Weight: 24×1 double

Values:

Min	119
Median	137
Max	194

Smoker: 24×1 logical

Values:

True	8
False	16

Systolic: 24×1 double

Values:

Min	114
Median	121.5
Max	138

Diastolic: 24×1 double

Values:

Min	68
Median	81.5
Max	96

SelfAssessedHealthStatus: 24×1 categorical

Values:

Excellent	7
Fair	3
Good	10
Poor	4

Delete the patients database table using the execute function.

```
sqlquery = ['DROP TABLE ' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Import Data from SQL Query Using Import Options

Customize import options when importing data from the results of an SQL query on a database. Control the import options by creating an `SQLImportOptions` object. Then, customize import options for different columns in the SQL query. Import data using the `fetch` function.

This example uses the `employees_database.mat` file, which contains the columns `first_name`, `hire_date`, and `DEPARTMENT_NAME`. The example also uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank username and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load employee information into the MATLAB® workspace.

```
employeedata = load('employees_database.mat');
```

Create the `employees` and `departments` database tables using the employee information.

```
emps = employeedata.employees;
depts = employeedata.departments;
```

```
sqlwrite(conn, 'employees', emps)
sqlwrite(conn, 'departments', depts)
```

Create an `SQLImportOptions` object using an SQL query and the `databaseImportOptions` function. This query retrieves all information for employees who are sales managers or programmers.

```
sqlquery = strcat("SELECT * from employees e join departments d ", ...
    "on (e.department_id = d.department_id) WHERE ", ...
    "(job_id = 'IT_PROG' or job_id = 'SA_MAN')");
opts = databaseImportOptions(conn, sqlquery)
```

```
opts =
    SQLImportOptions with properties:

        ExcludeDuplicates: false
        VariableNamingRule: 'modify'

        VariableNames: {'employee_id', 'first_name', 'last_name' ... and 13 more}
        VariableTypes: {'double', 'char', 'char' ... and 13 more}
        SelectedVariableNames: {'employee_id', 'first_name', 'last_name' ... and 13 more}
        FillValues: { NaN, '', '' ... and 13 more }
        RowFilter: <unconstrained>

        VariableOptions: Show all 16 VariableOptions
```

Display the current import options for the variables selected in the `SelectedVariableNames` property of the `SQLImportOptions` object.

```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts, vars)
```

```
varOpts =
    1x16 SQLVariableImportOptions array with properties:
```

Variable Options:

	(1)	(2)	(3)	(4)	(5)	(6)
Name:	'employee_id'	'first_name'	'last_name'	'email'	'phone_number'	'hire_date'
Type:	'double'	'char'	'char'	'char'	'char'	'char'
MissingRule:	'fill'	'fill'	'fill'	'fill'	'fill'	'fill'
FillValue:	NaN	''	''	''	''	''

To access sub-properties of each variable, use `getoptions`

Change the data types for the `hire_date`, `DEPARTMENT_NAME`, and `first_name` variables using the `setoptions` function. Then, display the updated import options. Because `hire_date` stores date and time data, change the data type of this variable to `datetime`. Because `DEPARTMENT_NAME` designates a finite set of repeating values, change the data type of this variable to `categorical`. Also, change the name of this variable to lowercase. Because `first_name` stores text data, change the data type of this variable to `string`.

```
opts = setoptions(opts, 'hire_date', 'Type', 'datetime');
opts = setoptions(opts, 'DEPARTMENT_NAME', 'Name', 'department_name', ...
    'Type', 'categorical');
opts = setoptions(opts, 'first_name', 'Type', 'string');
```



```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)
```

```
varOpts =
    1x16 SQLVariableImportOptions array with properties:
```

```
Variable Options:
```

	(1)	(2)	(3)	(4)	(5)	(6)
Name:	'employee_id'	'first_name'	'last_name'	'email'	'phone_number'	'hire_date'
Type:	'double'	'string'	'char'	'char'	'char'	'datetime'
MissingRule:	'fill'	'fill'	'fill'	'fill'	'fill'	'fill'
FillValue:	NaN	<missing>	'	'	'	'

To access sub-properties of each variable, use `getoptions`

Select the three modified variables using the `SelectVariableNames` property.

```
opts.SelectedVariableNames = ["first_name","hire_date","department_name"];
```

Set the filter condition to import only the data for the employees hired before January 1, 2006.

```
opts.RowFilter = opts.RowFilter.hire_date < datetime(2006,01,01)
```

```
opts =
    SQLImportOptions with properties:
```

```
ExcludeDuplicates: false
VariableNamingRule: 'modify'

VariableNames: {'employee_id', 'first_name', 'last_name' ... and 13 more}
VariableTypes: {'double', 'string', 'char' ... and 13 more}
SelectedVariableNames: {'first_name', 'hire_date', 'department_name'}
FillValues: { NaN, <missing>, '' ... and 13 more }
RowFilter: hire_date < 01-Jan-2006
```

```
VariableOptions: Show all 16 VariableOptions
```

Import and display the results of the SQL query using the `fetch` function.

```
employees_data = fetch(conn,sqlquery,opts)
```

```
employees_data=4x3 table
    first_name    hire_date    department_name
    _____    _____    _____
    "David"       25-Jun-2005    IT
    "John"        01-Oct-2004    Sales
    "Karen"       05-Jan-2005    Sales
    "Alberto"    10-Mar-2005    Sales
```

Delete the `employees` and `departments` database tables using the `execute` function.

```
execute(conn,'DROP TABLE employees')
execute(conn,'DROP TABLE departments')
```

Close the database connection.

```
close(conn)
```

Import Data from Database Table Using Import Options with Specified Catalog and Schema

Customize import options when importing data from a database table. Control the import options by creating an `SQLImportOptions` object. Specify the location of the database table by using the database catalog and schema. Then, customize import options for different database columns. Import data using the `sqlread` function.

This example uses the `patients.xls` file, which contains the columns `Gender`, `Location`, `SelfAssessedHealthStatus`, and `Smoker`. The example also uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table in the `toy_store` database catalog and `dbo` database schema using the patient information.

```
tablename = 'patients';
sqlwrite(conn,tablename,patients, ...
    'Catalog','toy_store','Schema','dbo')
```

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function. Specify the `toy_store` database catalog and `dbo` database schema for the location of the database table.

```
opts = databaseImportOptions(conn,tablename, ...
    'Catalog','toy_store','Schema','dbo');
```

Display the current import options for the variables selected in the `SelectedVariableNames` property of the `SQLImportOptions` object.

```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)
```

```
varOpts =
    1x10 SQLVariableImportOptions array with properties:
```

```
Variable Options:
      (1) |      (2) |      (3) |      (4) |      (5) |      (6) |      (7) |
Name: 'LastName' | 'Gender' | 'Age' | 'Location' | 'Height' | 'Weight' | 'Smoker' |
Type: 'char' | 'char' | 'double' | 'char' | 'double' | 'double' | 'double' |
MissingRule: 'fill' | 'fill' | 'fill' | 'fill' | 'fill' | 'fill' | 'fill' |
FillValue: '' | '' | NaN | '' | NaN | NaN | NaN |
```

To access sub-properties of each variable, use `getoptions`

Change the data types for the `Gender`, `Location`, `SelfAssessedHealthStatus`, and `Smoker` variables using the `setoptions` function. Because the `Gender`, `Location`, and

`SelfAssessedHealthStatus` variables indicate a finite set of repeating values, change their data type to categorical. Because the `Smoker` variable stores the values 0 and 1, change its data type to logical. Then, display the updated import options.

```
opts = setoptions(opts,{'Gender','Location','SelfAssessedHealthStatus'}, ...
    'Type','categorical');
opts = setoptions(opts,'Smoker','Type','logical');

varOpts = getoptions(opts,{'Gender','Location','Smoker', ...
    'SelfAssessedHealthStatus'})
```

```
varOpts =
    1x4 SQLVariableImportOptions array with properties:
```

Variable Options:

	(1)	(2)	(3)	(4)
Name:	'Gender'	'Location'	'Smoker'	'SelfAssessedHealthStatus'
Type:	'categorical'	'categorical'	'logical'	'categorical'
MissingRule:	'fill'	'fill'	'fill'	'fill'
FillValue:	<undefined>	<undefined>	0	<undefined>

To access sub-properties of each variable, use `getoptions`

Import the `patients` database table using the `sqlread` function, and display the last eight rows of the table.

```
data = sqlread(conn,tablename,opts,'Catalog','toy_store','Schema','dbo');
tail(data)
```

ans=8x10 table

LastName	Gender	Age	Location	Height	Weight	Smoker
{'Foster' }	Female	30	St. Mary's Medical Center	70	124	false
{'Gonzales' }	Male	48	County General Hospital	71	174	false
{'Bryant' }	Female	48	County General Hospital	66	134	false
{'Alexander' }	Male	25	County General Hospital	69	171	true
{'Russell' }	Male	44	VA Hospital	69	188	true
{'Griffin' }	Male	49	County General Hospital	70	186	false
{'Diaz' }	Male	45	County General Hospital	68	172	true
{'Hayes' }	Male	48	County General Hospital	66	177	false

Display a summary of the imported data. The `sqlread` function applies the import options to the variables in the imported data.

```
summary(data)
```

Variables:

LastName: 100x1 cell array of character vectors

Gender: 100x1 categorical

Values:

Female	53
Male	47

Age: 100×1 double

Values:

Min	25
Median	39
Max	50

Location: 100×1 categorical

Values:

County General Hospital	39
St. Mary s Medical Center	24
VA Hospital	37

Height: 100×1 double

Values:

Min	60
Median	67
Max	72

Weight: 100×1 double

Values:

Min	111
Median	142.5
Max	202

Smoker: 100×1 logical

Values:

True	34
False	66

Systolic: 100×1 double

Values:

Min	109
Median	122
Max	138

Diastolic: 100×1 double

Values:

Min	68
Median	81.5
Max	99

SelfAssessedHealthStatus: 100×1 categorical

Values:

Excellent	34
Fair	15
Good	40
Poor	11

Delete the `patients` database table from the `toy_store` database catalog and the `dbo` database schema by using the `execute` function.

```
sqlquery = ['DROP TABLE toy_store.dbo.' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with the `database` function, connection object created with the `mysql` function, connection object created with the `postgresql` function, or `sqlite` object.

source — Source

character vector | string scalar

Source, specified as a character vector or string scalar. Use the `source` input argument to specify the name of a database table or an SQL query for importing data from a database.

Example: "inventorytable"

Example: "SELECT * FROM inventorytable"

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . ,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `opts =`

```
databaseImportOptions(conn,"inventorytable",'Catalog',"toystore_doc",'Schema',
,"dbo") defines import options for importing data from the inventorytable database table
located in the toystore_doc catalog and dbo schema.
```

Catalog — Database catalog name

character vector | string scalar

Database catalog name, specified as the comma-separated pair consisting of 'Catalog' and a character vector or string scalar. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have numerous catalogs.

Use the 'Catalog' name-value pair argument only when source is a database table.

Example: 'Catalog', 'toy_store'

Data Types: char | string

Schema — Database schema name

character vector | string scalar

Database schema name, specified as the comma-separated pair consisting of 'Schema' and a character vector or string scalar. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Use the 'Schema' name-value pair argument only when source is a database table.

Example: 'Schema', 'dbo'

Data Types: char | string

Output Arguments

opts — Database import options

SQLImportOptions object

Database import options, returned as an SQLImportOptions object.

Version History

Introduced in R2018b

See Also

setoptions | getoptions | reset | close | database | execute | sqlwrite | sqlread | fetch | mysql | postgresql

Topics

“Customize Options for Importing Data from Database into MATLAB” on page 5-67

“Importing Data Common Errors” on page 3-3

External Websites

SQL Tutorial

getoptions

Package: database.options

Retrieve import options for database data

Syntax

```
varOpts = getoptions(opts)
varOpts = getoptions(opts,varnames)
varOpts = getoptions(opts,index)
```

Description

`varOpts = getoptions(opts)` returns the import options for all variables in the `SQLImportOptions` object.

`varOpts = getoptions(opts,varnames)` returns the import options for the specified variable names.

`varOpts = getoptions(opts,index)` returns the import options for the variables specified by a numeric index.

Examples

Retrieve Default Import Options for Database Table

Control the import options by creating an `SQLImportOptions` object. Then, retrieve the default import options from a database table.

This example uses the `patients.xls` spreadsheet, which contains patient information. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';
sqlwrite(conn,tablename,patients)
```

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename);
```

Retrieve and display the default import options for the `patients` database table.

```
varOpts = getoptions(opts)
```

```
varOpts =  
    1x10 SQLVariableImportOptions array with properties:
```

```
    Variable Options:
```

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Name:	'LastName'	'Gender'	'Age'	'Location'	'Height'	'Weight'	'Smoker'	'SystolicBP'	'DiastolicBP'	'Cholesterol'
Type:	'char'	'char'	'double'	'char'	'double'	'double'	'double'	'double'	'double'	'double'
FillValue:	''	''	[NaN]	''	[NaN]	[NaN]	[NaN]	[NaN]	[NaN]	[NaN]

To access sub-properties of each variable, use `getoptions`

To modify the variable import options, see the `setoptions` function.

Delete the `patients` database table using the `execute` function.

```
sqlquery = ['DROP TABLE ' tablename];  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Retrieve Default Import Options for Database Columns Using Variable Names

Control the import options by creating an `SQLImportOptions` object. Then, retrieve the default import options for several columns from a database table. Specify the columns to retrieve by using the database column names.

This example uses the `patients.xls` spreadsheet, which contains the columns `LastName`, `Age`, and `Location`. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';  
sqlwrite(conn,tablename,patients)
```

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.


```
opts = databaseImportOptions(conn,tablename);
```

Specify the names of the database columns in the `patients` database table.

```
varnames = {'LastName','Age','Location'};
```

Retrieve and display the default import options for the specified database columns.

```
varOpts = getoptions(opts,varnames)
```

```
varOpts =
    1x3 SQLVariableImportOptions array with properties:
```

```
Variable Options:
      (1) |      (2) |      (3)
Name: 'LastName' | 'Age' | 'Location'
Type: 'char' | 'double' | 'char'
FillValue: '' | [NaN] | ''
```

To access sub-properties of each variable, use `getoptions`

To modify the variable import options, see the `setoptions` function.

Delete the `patients` database table using the `execute` function.

```
sqlquery = ['DROP TABLE ' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Retrieve Default Import Options for Database Columns Using Numeric Index

Control the import options by creating an `SQLImportOptions` object. Then, retrieve the default import options for several columns from a database table. Specify the columns to retrieve by using a numeric index.

This example uses the `patients.xls` spreadsheet, which contains the columns `LastName`, `Gender`, and `Age`. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';
sqlwrite(conn,tablename,patients)
```

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename);
```

Specify the first three database columns by using a numeric index.

```
index = [1,2,3];
```

Retrieve and display the default import options for the specified database columns.

```
varOpts = getoptions(opts,index)
```

```
varOpts =
    1x3 SQLVariableImportOptions array with properties:

    Variable Options:
           (1) |           (2) |           (3)
    Name: 'LastName' | 'Gender' | 'Age'
    Type: 'char' | 'char' | 'double'
    FillValue: '' | '' | [NaN]
```

To access sub-properties of each variable, use `getoptions`

To modify the variable import options, see the `setoptions` function.

Delete the `patients` database table using the `execute` function.

```
sqlquery = ['DROP TABLE ' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

opts — Database import options

`SQLImportOptions` object

Database import options, specified as an `SQLImportOptions` object.

varnames — Variable names

character vector | cell array of character vectors | string scalar | string array | numeric vector

Variable names, specified as a character vector, cell array of character vectors, string scalar, string array, or numeric vector. The `varnames` input argument indicates the variables in the `VariableNames` property of the `SQLImportOptions` object to use for importing data.

Example: 'productname'

Data Types: double | char | string | cell

index — Index

numeric vector

Index, specified as a numeric vector that identifies the variables in the `VariableNames` property of the `SQLImportOptions` object to use for importing data.

Example: `[1,2,3]`

Data Types: `double`

Output Arguments**varOpts — Type-dependent options for selected variables**

array of variable import options objects

Type-dependent options for selected variables, returned as an array of variable import options objects. The array contains an object corresponding to each variable in the `opts` input argument or in the selected variables specified by the `varnames` or `index` input argument. The data type of each object in the array depends on the data type of the corresponding variable.

For `categorical` and `datetime` data types, each variable import options object contains additional properties that correspond to the data type.

To modify the properties of the individual objects, use the `setoptions` function.

Version History**Introduced in R2018b****See Also**

`databaseImportOptions` | `setoptions` | `reset` | `close` | `database` | `execute` | `sqlwrite` | `sqlread`

Topics

“Customize Options for Importing Data from Database into MATLAB” on page 5-67

“Importing Data Common Errors” on page 3-3

External Websites

SQL Tutorial

preview

Package: database.options

Preview eight rows from database using import options

Syntax

```
T = preview(opts)
```

Description

`T = preview(opts)` returns a table containing the first eight rows of database data by using the `SQLImportOptions` object. The value of the `SelectedVariableNames` property of the `SQLImportOptions` object specifies the variables that appear in the table.

Usually, the table contains eight rows of data. However, in some instances, the number of rows differs depending on property values defined in the `SQLImportOptions` object. The `preview` function returns fewer than eight rows if:

- The SQL query or table contains fewer than eight rows of data.
- The SQL query or table is empty or the `MissingRule` import option (of the variable import options) specifies to omit rows that contain missing data. To access the values of variable import options, use the `getoptions` function.

Examples

Preview Database Data After Customizing Options

Customize import options when importing text data from a database table. Control the import options by creating an `SQLImportOptions` object. Customize the import options for a text database column. Preview the database data before importing data. Then, import the data using the `sqlread` function.

This example uses the `patients.xls` spreadsheet, which contains the first column `LastName`. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';  
sqlwrite(conn, tablename, patients)
```

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename);
```

Retrieve the default import options for the `LastName` variable.

```
varnames = 'LastName';
varOpts = getoptions(opts,varnames)

varOpts =
  SQLVariableImportOptions with properties:

  Variable Properties :
      Name: 'LastName'
      Type: 'char'
      MissingRule: 'fill'
      FillValue: ''

  String Properties :
      WhitespaceRule: 'preserve'
      TextCaseRule: 'preserve'
```

Set the import options for the data type of the `LastName` variable to `string`. Specify the `LastName` variable by using a numeric index that finds the variable within the `SelectedVariables` property of the `SQLImportOptions` object. Also, set the import options to replace missing data in the `LastName` variable with the `NoName` fill value.

```
index = 1;
opts = setoptions(opts,index,'Type','string', ...
  'FillValue','NoName');
```

Preview the first eight rows of database data using the import options. The data preview shows that the `LastName` variable has the `string` data type.

```
T = preview(opts)
```

```
T=8x10 table
  LastName      Gender      Age      Location      Height      Weight      Smoker
  _____  _____  _____  _____  _____  _____  _____
  "Smith"      'Male'      38      'County General Hospital'      71      176      1
  "Johnson"    'Male'      43      'VA Hospital'      69      163      0
  "Williams"    'Female'    38      'St. Mary's Medical Center'      64      131      0
  "Jones"       'Female'    40      'VA Hospital'      67      133      0
  "Brown"       'Female'    49      'County General Hospital'      64      119      0
  "Davis"       'Female'    46      'St. Mary's Medical Center'      68      142      0
  "Miller"      'Female'    33      'VA Hospital'      64      142      1
  "Wilson"      'Male'      40      'VA Hospital'      68      180      0
```

Import the text data in the selected variable and display the first eight rows. The imported data shows that the variable has the `string` data type.

```
opts.SelectedVariableNames = 'LastName';
data = sqlread(conn,tablename,opts);
head(data)
```

```
ans=8x1 table
  LastName
  _____
  "Smith"
  "Johnson"
  "Williams"
  "Jones"
  "Brown"
  "Davis"
  "Miller"
  "Wilson"
```

Delete the patients database table using the execute function.

```
sqlquery = ['DROP TABLE ' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

opts — Database import options

SQLImportOptions object

Database import options, specified as an SQLImportOptions object.

Version History

Introduced in R2019a

See Also

databaseImportOptions | getoptions | setoptions | reset | close | database

Topics

“Customize Options for Importing Data from Database into MATLAB” on page 5-67

“Importing Data Common Errors” on page 3-3

External Websites

SQL Tutorial

reset

Package: database.options

Reset to default import options for database data

Syntax

```
opts = reset(opts)
```

Description

`opts = reset(opts)` resets the import options for importing data from a database back to the original state. The function returns the `SQLImportOptions` object. The `VariableNames`, `VariableTypes`, and `FillValues` properties of the `SQLImportOptions` object revert to the default values.

Examples

Reset Options When Importing Data

Reset import options when importing numeric data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options for a numeric database column. Import data using the `sqlread` function. Then, reset the import options back to the original state.

This example uses the `patients.xls` file, which contains the column `Weight`. The example also uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create an ODBC database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';  
sqlwrite(conn, tablename, patients)
```

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn, tablename);
```

Retrieve the import options for the `Weight` variable. This variable has the `double` data type.

```
varnames = 'Weight';
varOpts = getoptions(opts,varnames)

varOpts =
  SQLVariableImportOptions with properties:

  Variable Properties :
      Name: 'Weight'
      Type: 'double'
  FillValue: NaN
```

Customize the import options for the `Weight` column in the `patients` database table. Because this column contains numeric data, change the data type to `int64`.

```
opts = setoptions(opts,varnames,'Type','int64');
```

Import the numeric data in the specified column and display a summary of the imported variable. The summary shows that the variable has the `int64` data type.

```
opts.SelectedVariableNames = varnames;
data = sqlread(conn,tablename,opts);
summary(data)
```

Variables:

```
Weight: 100×1 int64
```

Values:

Min	111
Median	143
Max	202

Reset the import options back to their original state, and retrieve the import options for the `Weight` variable. This variable has the `double` data type again.

```
opts = reset(opts);
varOpts = getoptions(opts,varnames)

varOpts =
  SQLVariableImportOptions with properties:

  Variable Properties :
      Name: 'Weight'
      Type: 'double'
  FillValue: NaN
```

Import the numeric data again using the default import options, and display a summary of the imported variable.

```
opts.SelectedVariableNames = varnames;
data = sqlread(conn,tablename,opts);
summary(data)
```

Variables:

```
Weight: 100×1 double
```


Values:

Min	111
Median	142.5
Max	202

Delete the `patients` database table using the `execute` function.

```
sqlquery = ['DROP TABLE ' tablename];  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

opts — Database import options

SQLImportOptions object

Database import options, specified as an SQLImportOptions object.

Version History

Introduced in R2018b

See Also

[databaseImportOptions](#) | [setoptions](#) | [getoptions](#) | [close](#) | [database](#) | [execute](#) | [sqlwrite](#)

Topics

“Customize Options for Importing Data from Database into MATLAB” on page 5-67

“Importing Data Common Errors” on page 3-3

External Websites

SQL Tutorial

setoptions

Package: database.options

Customize import options for database data

Syntax

```
opts = setoptions(opts,varnames,  
Option1,OptionValue1,...,OptionN,OptionValueN)  
opts = setoptions(opts,index,Option1,OptionValue1,...,OptionN,OptionValueN)
```

Description

`opts = setoptions(opts,varnames,Option1,OptionValue1,...,OptionN,OptionValueN)` customizes the import options for importing data from a database into MATLAB. The function returns the `SQLImportOptions` object. To import data, you use the `SQLImportOptions` object, the specified variable names, and the import options with their corresponding values.

`opts = setoptions(opts,index,Option1,OptionValue1,...,OptionN,OptionValueN)` customizes the import options for the variables specified by a numeric index.

Examples

Customize Options When Importing Numeric Data

Customize import options when importing numeric data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options for a numeric database column. Import data using the `sqlread` function.

This example uses the `patients.xls` spreadsheet, which contains the column `Weight`. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';  
conn = database(datasource, '', '');
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';  
sqlwrite(conn,tablename,patients)
```

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename);
```

Customize the import options for the `Weight` column in the `patients` database table. Because this column is numeric, change the data type to `int64`.

```
varnames = 'Weight';
opts = setoptions(opts,varnames,'Type','int64');
```

Import the numeric data in the specified column and display a summary of the imported variable. The summary shows that the variable has the `int64` data type.

```
opts.SelectedVariableNames = {'Weight'};
data = sqlread(conn,tablename,opts);
summary(data)
```

Variables:

```
Weight: 100x1 int64
```

Values:

Min	111
Median	143
Max	202

Delete the `patients` database table using the `execute` function.

```
sqlquery = ['DROP TABLE ' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Customize Options When Importing Text Data

Customize import options when importing text data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options for a text database column. Import data using the `sqlread` function.

This example uses the `patients.xls` spreadsheet, which contains the first column `LastName`. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MS SQL Server Auth";
conn = database(datasource,"","");
```

Load patient information into the MATLAB® workspace.

```
patients = readtable("patients.xls");
```

Create the `patients` database table using the patient information.

```
tablename = "patients";
sqlwrite(conn,tablename,patients)
```

Create an `SQLImportOptions` object using the `patients` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename);
```

Retrieve the default import options for the `LastName` and `SelfAssessedHealthStatus` variables.

```
varnames = ["LastName" "SelfAssessedHealthStatus"];
varOpts = getoptions(opts,varnames)

varOpts =
    1x2 SQLVariableImportOptions array with properties:
```

```
Variable Options:
      (1) | (2)
      Name: 'LastName' | 'SelfAssessedHealthStatus'
      Type: 'char' | 'char'
MissingRule: 'fill' | 'fill'
      FillValue: '' | ''
```

To access sub-properties of each variable, use `getoptions`

Set the import options for the data type of the `LastName` variable to `string`. Specify the `LastName` variable by using a numeric index that finds the variable within the `SelectedVariables` property of the `SQLImportOptions` object. Also, set the import options to replace missing data in the `LastName` variable with the `NoName` fill value.

```
index = 1;
opts = setoptions(opts,index,'Type',"string", ...
    'FillValue',"NoName");
```

Set the import options for the text case of the `SelfAssessedHealthStatus` variable to uppercase.

```
varname = "SelfAssessedHealthStatus";
opts = setoptions(opts,varname,'TextCaseRule',"upper");
```

Import the text data in the selected variables and display the first eight rows. The imported data shows that the `LastName` variable has the `string` data type and the `SelfAssessedHealthStatus` variable text is uppercase.

```
opts.SelectedVariableNames = ["LastName" "SelfAssessedHealthStatus"];
T = sqlread(conn,tablename,opts);
head(T)
```

```
ans=8x2 table
      LastName      SelfAssessedHealthStatus
      _____      _____
      "Smith"          'EXCELLENT'
      "Johnson"       'FAIR'
      "Williams"      'GOOD'
      "Jones"          'FAIR'
```

```
"Brown"          'GOOD'
"Davis"          'GOOD'
"Miller"        'GOOD'
"Wilson"        'GOOD'
```

Delete the `patients` database table using the `execute` function.

```
sqlstr = "DROP TABLE ";
sqlquery = strcat(sqlstr,tablename);
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Customize Options When Importing Date and Time Data

Customize import options when importing date and time data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options for database columns that contain date and time data. Import data using the `sqlread` function.

This example uses the `outages.csv` file, which contains the columns `OutageTime` and `RestorationTime`. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load outage information into the MATLAB® workspace.

```
outages = readtable('outages.csv');
```

Create the `outages` database table using the outage information.

```
tablename = 'outages';
sqlwrite(conn,tablename,outages)
```

Retrieve the data using the `sqlread` function and display the first eight rows. The second row of the `RestorationTime` variable contains missing data.

```
data = sqlread(conn,tablename);
head(data)
```

ans=8×6 table

Region	OutageTime	Loss	Customers	RestorationTime
'SouthWest'	'2002-02-01 12:18:00.000'	458.98	1.8202e+06	'2002-02-07 16:50:00.000'
'SouthEast'	'2003-01-23 00:49:00.000'	530.14	2.1204e+05	'
'SouthEast'	'2003-02-07 21:15:00.000'	289.4	1.4294e+05	'2003-02-17 08:14:00.000'
'West'	'2004-04-06 05:44:00.000'	434.81	3.4037e+05	'2004-04-06 06:10:00.000'

```

'MidWest'      '2002-03-16 06:18:00.000'  186.44  2.1275e+05  '2002-03-18 23:23:00.000
'West'         '2003-06-18 02:49:00.000'      0        0          '2003-06-18 10:54:00.000
'West'         '2004-06-20 14:39:00.000'    231.29   NaN        '2004-06-20 19:16:00.000
'West'         '2002-06-06 19:28:00.000'    311.86   NaN        '2002-06-07 00:51:00.000

```

Create an `SQLImportOptions` object using the `outages` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename);
```

Retrieve the default import options for the `OutageTime` and `RestorationTime` variables.

```
varnames = {'OutageTime','RestorationTime'};
varOpts = getoptions(opts,varnames)
```

```
varOpts =
    1x2 SQLVariableImportOptions array with properties:
```

```

Variable Options:
              (1) |              (2)
      Name: 'OutageTime' | 'RestorationTime'
      Type:   'char' |   'char'
      FillValue: '' | ''

```

To access sub-properties of each variable, use `getoptions`

Set the import options for the data type of the specified variables to `datetime`. Also, set the import options to replace missing data in the specified variables with the current date and time.

```
opts = setoptions(opts,varnames,'Type','datetime', ...
    'FillValue',datetime('now'));
```

Import the date and time data in the selected variables and display the first eight rows. The imported data shows that the variables have the `datetime` data type. The missing value in the second row of the `RestorationTime` variable is filled with the current date and time.

```
opts.SelectedVariableNames = varnames;
T = sqlread(conn,tablename,opts);
head(T)
```

```
ans=8x2 table
      OutageTime      RestorationTime
-----
01-Feb-2002 12:18:00 07-Feb-2002 16:50:00
23-Jan-2003 00:49:00 19-Jun-2018 15:30:14
07-Feb-2003 21:15:00 17-Feb-2003 08:14:00
06-Apr-2004 05:44:00 06-Apr-2004 06:10:00
16-Mar-2002 06:18:00 18-Mar-2002 23:23:00
18-Jun-2003 02:49:00 18-Jun-2003 10:54:00
20-Jun-2004 14:39:00 20-Jun-2004 19:16:00
06-Jun-2002 19:28:00 07-Jun-2002 00:51:00

```

Delete the `outages` database table using the `execute` function.

```
sqlquery = ['DROP TABLE ' tablename];
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Customize Options When Importing Categorical Array Data

Customize import options when importing categorical array data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options for database columns that contain categorical array data. Import data using the `sqlread` function.

This example uses the `outages.csv` file, which contains the columns `Region` and `Cause`. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load outage information into the MATLAB® workspace.

```
outages = readtable('outages.csv');
```

Create the `outages` database table using the outage information.

```
tablename = 'outages';
sqlwrite(conn,tablename,outages)
```

Create an `SQLImportOptions` object using the `outages` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename);
```

Retrieve the default import options for the `Region` and `Cause` variables.

```
varnames = {'Region','Cause'};
varOpts = getoptions(opts,varnames)
```

```
varOpts =
    1x2 SQLVariableImportOptions array with properties:
```

```
Variable Options:
      (1) | (2)
Name: 'Region' | 'Cause'
Type: 'char' | 'char'
FillValue: '' | ''
```

To access sub-properties of each variable, use `getoptions`

Set the import options for the data type of the specified variables to `categorical`. Also, set the import options to replace missing data in the specified variables with the fill value `unknown`.

```
opts = setoptions(opts,varnames,'Type','categorical', ...  
    'FillValue','unknown');
```

Import the categorical array data in the selected variables and display a summary of the data. The imported data shows that the variables have the `categorical` data type. The missing values of both variables are filled with the value `unknown`.

```
opts.SelectedVariableNames = varnames;  
T = sqlread(conn,tablename,opts);  
summary(T)
```

Variables:

Region: 1468×1 categorical

Values:

MidWest	142
NorthEast	557
SouthEast	389
SouthWest	26
West	354
unknown	0

Cause: 1468×1 categorical

Values:

attack	294
earthquake	2
energy emergency	188
equipment fault	156
fire	25
severe storm	338
thunder storm	201
unknown	24
wind	95
winter storm	145

Delete the `outages` database table using the `execute` function.

```
sqlquery = ['DROP TABLE ' tablename];  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Customize Options When Importing Logical Data

Customize import options when importing logical data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options for database columns that contain logical data. Import data using the `sqlread` function.

This example uses the `airlinesmall_subset.xls` spreadsheet, which contains the column `Cancelled`. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = 'MS SQL Server Auth';
conn = database(datasource, '', '');
```

Load flight information in the MATLAB® workspace.

```
flights = readtable('airlinesmall_subset.xlsx');
```

Create the `flights` database table using the flight information.

```
tablename = 'flights';
sqlwrite(conn, tablename, flights)
```

Create an `SQLImportOptions` object using the `flights` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn, 'flights');
```

Retrieve the default import options for the `Cancelled` variable.

```
varnames = 'Cancelled';
varOpts = getoptions(opts, varnames)

varOpts =
  SQLVariableImportOptions with properties:
```

```
  Variable Properties :
      Name: 'Cancelled'
      Type: 'double'
  FillValue: NaN
```

Set the import options for the data type of the specified variable to `logical`. Also, set the import options to replace missing data in the specified variable with the fill value `true`.

```
opts = setoptions(opts, varnames, 'Type', 'logical', ...
  'FillValue', true);
```

Import the logical data in the selected variable and display a summary of the data. The imported data shows that the variable has the `logical` data type.

```
opts.SelectedVariableNames = varnames;
T = sqlread(conn, tablename, opts);
summary(T)
```

Variables:

```
Cancelled: 1338×1 logical
```

Values:

```
True      29
False    1309
```

Delete the `flights` database table using the `execute` function.

```
sqlquery = 'DROP TABLE flights';  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Customize Options to Omit Missing Data

Customize import options when importing data from a database table. Control the import options by creating an `SQLImportOptions` object. Then, customize the import options to omit missing data. Import data using the `sqlread` function.

This example uses the `outages.csv` file, which contains outage data. Also, the example uses a Microsoft® SQL Server® Version 11.00.2100 database and the Microsoft SQL Server Driver 11.00.5058.

Create a database connection to a Microsoft SQL Server database with Windows® authentication. Specify a blank user name and password.

```
datasource = "MS SQL Server Auth";  
conn = database(datasource, "", "");
```

Load outage information into the MATLAB® workspace.

```
outages = readtable("outages.csv");
```

Create the `outages` database table using the outage information.

```
tablename = "outages";  
sqlwrite(conn,tablename,outages)
```

Create an `SQLImportOptions` object using the `outages` database table and the `databaseImportOptions` function.

```
opts = databaseImportOptions(conn,tablename);
```

Determine the size of `outages`.

```
size(outages)
```

```
ans = 1×2
```

```
1468      6
```

Set the import options to omit rows that have missing data in the `Customers` variable.

```
varnames = "Customers";  
opts = setoptions(opts,varnames,'MissingRule','omitrow');
```

Import the data and display the first eight rows. The imported data contains no missing data in the `Customers` variable.

```
T = sqlread(conn,tablename,opts);
head(T)
```

```
ans=8x6 table
```

Region	OutageTime	Loss	Customers	RestorationTime
'SouthWest'	'2002-02-01 12:18:00.000'	458.98	1.8202e+06	'2002-02-07 16:50:00.000'
'SouthEast'	'2003-01-23 00:49:00.000'	530.14	2.1204e+05	'
'SouthEast'	'2003-02-07 21:15:00.000'	289.4	1.4294e+05	'2003-02-17 08:14:00.000'
'West'	'2004-04-06 05:44:00.000'	434.81	3.4037e+05	'2004-04-06 06:10:00.000'
'MidWest'	'2002-03-16 06:18:00.000'	186.44	2.1275e+05	'2002-03-18 23:23:00.000'
'West'	'2003-06-18 02:49:00.000'	0	0	'2003-06-18 10:54:00.000'
'NorthEast'	'2003-07-16 16:23:00.000'	239.93	49434	'2003-07-17 01:12:00.000'
'MidWest'	'2004-09-27 11:09:00.000'	286.72	66104	'2004-09-27 16:37:00.000'

Determine the size of T. The number of rows in the imported data is smaller because the software removes all rows with missing data in the `Customers` variable.

```
size(T)
```

```
ans = 1x2
```

```
1140      6
```

Delete the outages database table using the `execute` function.

```
sqlstr = "DROP TABLE ";
sqlquery = strcat(sqlstr,tablename);
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

opts — Database import options

SQLImportOptions object

Database import options, specified as an SQLImportOptions object.

varnames — Variable names

character vector | cell array of character vectors | string scalar | string array | numeric vector

Variable names, specified as a character vector, cell array of character vectors, string scalar, string array, or numeric vector. The `varnames` input argument indicates the variables in the `VariableNames` property of the SQLImportOptions object to use for importing data.

Example: 'productname'

Data Types: double | char | string | cell

index — Index

numeric vector

Index, specified as a numeric vector that identifies the variables in the `VariableNames` property of the `SQLImportOptions` object to use for importing data.

Example: `[1,2,3]`

Data Types: `double`

Option1,OptionValue1,...,OptionN,OptionValueN – Import options

name-value pair arguments

Import options, specified as one or more name-value pair arguments. `Option` is a character vector or string scalar that specifies the name of an import option. `OptionValue` specifies the value of the import option.

Example: `'FillValue',true,'Type','logical'` sets the data type of the specified variable as `logical` and sets the fill value for missing data in the specified variable as `true`.

Example: `'Name','Location'` changes the name of the specified variable to `Location`.

All Variables

You can set import options to change the value of missing data, the name of a variable, or the data type of a variable. These import options apply to all variables specified by either the `varnames` or `index` input argument.

Import Option Name	Description	Import Option Values
'FillValue'	Missing data value	<p>Value must be a scalar for a single variable or a cell array for multiple variables.</p> <p>Valid data types are:</p> <ul style="list-style-type: none"> • All integer classes • <code>single</code> • <code>double</code> • <code>logical</code> • <code>char</code> • <code>string</code> • <code>datetime</code> • <code>missing</code> <p>The data type depends on the variable type in the database.</p>

Import Option Name	Description	Import Option Values
'MissingRule'	Missing data rule	<p>Value must be one of the following:</p> <ul style="list-style-type: none"> 'fill' (default) — Replace missing data with the missing data value specified by the 'FillValue' import option. 'omitrow' — Omit rows that contain missing data. <p>You can specify these values as a character vector or string scalar.</p> <p>Setting the 'MissingRule' import option is the equivalent of using the IS NOT NULL SQL statement in ANSI SQL.</p>
'Name'	Variable name	Value must be a character vector or string scalar for a single variable or a cell array of character vectors or string array for multiple variables.
'Type'	Data type	Value must be a character vector or string scalar for a single variable or a cell array of character vectors or string array for multiple variables.

The following table describes the valid import option values for the 'Type' import option. The first column shows the data types in the `VariableTypes` property of the `SQLImportOptions` object. The second column shows the valid data types to specify in the character vector. To use the valid data type value, enclose it in quotes (for example, 'single').

Variable Data Type	Valid Data Type Values for 'Type' Import Option
<ul style="list-style-type: none"> • Any integer class • single • double 	<ul style="list-style-type: none"> • Any integer class • single • double • logical • categorical <p>The undefined floating-point numbers NaN, -Inf, and +Inf are valid only for the single and double data types. When you change the data type of a floating-point number to an integer, the function that imports the data from the database converts the undefined floating-point number. For example, when you change the data type to 'int8':</p> <ul style="list-style-type: none"> • NaN values change to 0. • -Inf values change to intmin('int8'). • +Inf values change to intmax('int8'). <p>For details, see intmin and intmax.</p> <p>The same conversion applies to all integer classes.</p>
logical	<ul style="list-style-type: none"> • All integer classes • single • double • logical • categorical
char or string	<ul style="list-style-type: none"> • char • string • datetime • duration • categorical <p>You can change the VariableTypes property to datetime only if the column in the database table has the DATETIME data type. Also, you can change the VariableTypes property to duration only if the column in the database table has the TIME data type.</p>
datetime	<ul style="list-style-type: none"> • char • string • datetime

Variable Data Type	Valid Data Type Values for 'Type' Import Option
duration	<ul style="list-style-type: none"> • char • string • duration
categorical	<ul style="list-style-type: none"> • All integer classes • single • double • char • string • logical • categorical

Variables with Text Data Type

You can set import options to change the value of variables with a text data type. These import options apply to variables that are either character vectors or string arrays specified by either the `varnames` or `index` input argument. You can specify the import option values as a character vector or string scalar.

Import Option Name	Description	Import Option Values
'WhiteSpaceRule'	Leading and trailing white spaces	<ul style="list-style-type: none"> • 'preserve' (default) — Preserve white spaces. • 'trim' — Remove any leading and trailing white spaces from the text. This import option value ignores white spaces in the middle of the text. • 'trimleading' — Remove only the leading white spaces. • 'trimtrailing' — Remove only the trailing white spaces.
'TextCaseRule'	Text case	<ul style="list-style-type: none"> • 'preserve' (default) — Preserve the text case. • 'lower' — Convert text to lowercase. • 'upper' — Convert text to uppercase.

Variables with datetime Data Type

You can set import options to change the value of variables with the `datetime` data type. These import options apply to variables with the `datetime` data type specified by either the `varnames` or `index` input argument.

Import Option Name	Description	Import Option Values	Default Import Option Value
'DateTimeFormat'	Display format of dates and times	For valid values, see the description of the <code>Format</code> property in the <code>datetime</code> function.	'default'
'DateTimeLocale'	Locale to use for interpreting dates	For valid values, see the description of the <code>'Locale'</code> name-value pair argument in the <code>datetime</code> function.	'en-US'
'TimeZone'	Time zone	For valid values, see the description of the <code>TimeZone</code> property in the <code>datetime</code> function.	' '
'InputFormat'	Format of the input text representing dates and times	For valid values, see the description of the <code>infmt</code> input argument in the <code>datetime</code> function.	'yyyy-MM-dd HH:mm:ss.SSSSSSSS'

Variables with duration Data Type

You can set import options to change the value of variables with the `duration` data type. These import options apply to variables with the `duration` data type specified by either the `varnames` or `index` input argument.

Import Option Name	Description	Import Option Values	Default Import Option Value
'InputFormat'	Format of the input text representing time	For valid values, see the description of the <code>infmt</code> input argument in the <code>duration</code> function.	' '
'DurationFormat'	Display format of time	For valid values, see the description of the <code>Format</code> property in the <code>duration</code> function.	'hh:mm:ss'

Variable with categorical Data Type

You can set import options to change the value of variables with the `categorical` data type. These import options apply to variables with the `categorical` data type specified by either the `varnames` or `index` input argument.

Import Option Name	Description	Import Option Values	Default Import Option Value
'Categories'	Expected categories	For valid values, see the description of the <code>catnames</code> input argument in the <code>categorical</code> function.	{}
'Protected'	Category protection indicator	For valid values, see the description of the 'Protected' name-value pair argument in the <code>categorical</code> function.	false
'Ordinal'	Mathematical ordering indicator	For valid values, see the description of the 'Ordinal' name-value pair argument in the <code>categorical</code> function.	false

Data Types: char | string

Version History

Introduced in R2018b

See Also

`databaseImportOptions` | `getoptions` | `preview` | `reset` | `close` | `database` | `execute` | `sqlwrite` | `sqlread`

Topics

“Customize Options for Importing Data from Database into MATLAB” on page 5-67

“Importing Data Common Errors” on page 3-3

External Websites

SQL Tutorial

connection

MySQL native interface database connection

Description

Create a connection to a MySQL database using the MySQL native interface. Configure a MySQL native interface data source using the `databaseConnectionOptions` function. For details, see “Configure MySQL Native Interface Data Source” on page 6-2.

Creation

Create a `connection` object by using the `mysql` function.

Properties

DataSource — Data source name

string scalar

This property is read-only.

Data source name, specified as a string scalar.

Example: "MySQLDataSource"

Data Types: string

Database — Database name

"" (default) | string scalar

This property is read-only.

Database name, specified as a string scalar.

If you use the 'DatabaseName' name-value pair argument of the `mysql` function, the `mysql` function sets the `Database` property of the `connection` object to the specified value.

Example: "toystore_doc"

Data Types: string

Server — Server name

localhost (default) | string scalar

This property is read-only.

Server name, specified as a string scalar.

If you use the 'Server' name-value pair argument of the `mysql` function, the `mysql` function sets the `Server` property of the `connection` object to the specified value.

Example: "dbtb00"

Data Types: `string`

UserName — User name

`""` (default) | `string` scalar

This property is read-only.

User name, specified as a `string` scalar.

Data Types: `string`

DefaultCatalog — Default catalog

`""` (default) | `string` scalar

This property is read-only.

Default catalog, specified as a `string` scalar.

Example: `"toy_store"`

Data Types: `string`

Catalogs — Catalogs in database

`""` (default) | `string` array

This property is read-only.

Catalogs in database, specified as a `string` array.

Example: `["information", "mysql"]`

Data Types: `string`

Schemas — Schemas in database

`""` (default) | `string` array

This property is read-only.

Schemas in database, specified as a `string` array.

Example: `["information_schema", "toys"]`

Data Types: `string`

AutoCommit — Flag to autocommit transactions

`string` scalar

Flag to autocommit transactions, specified as one of these values:

- `"on"` — Database transactions are automatically committed to the database.
- `"off"` — Database transactions must be committed to the database manually.

You can set this property by using dot notation.

LoginTimeout — Login timeout

`0` (default) | `positive numeric` scalar

This property is read-only.

Login timeout, specified as a positive numeric scalar. The login timeout specifies the number of seconds that the driver waits while trying to connect to a database before throwing an error.

When no login timeout for the connection attempt is specified, the value is 0.

When a login timeout is not supported by the database, the value is -1.

Data Types: `double`

MaxDatabaseConnections — Maximum number of database connections

-1 (default) | positive numeric scalar

This property is read-only.

Maximum number of database connections, specified as a positive numeric scalar.

When the database has no upper limit to the maximum number of database connections, the value is 0.

When a maximum number of database connections is not supported by the database, the value is -1.

Data Types: `double`

DatabaseProductName — Database product name

"" (default) | string scalar

This property is read-only.

Database product name, specified as a string scalar.

Example: "MySQL"

Data Types: `string`

DatabaseProductVersion — Database product version

"" (default) | string scalar

This property is read-only.

Database product version, specified as a string scalar.

Example: "5.7.22"

Data Types: `string`

DriverName — Driver name

"" (default) | string scalar

This property is read-only.

Driver name of the MySQL driver, specified as a string scalar.

Example: "Mariadb Connector/C"

Data Types: `string`

DriverVersion — Driver version

"" (default) | string scalar

This property is read-only.

Driver version of the MySQL driver, specified as a string scalar.

Example: "3.2.5"

Data Types: string

Object Functions

Manage MySQL Database Connection

`close` Close MySQL native interface database connection
`isopen` Determine if MySQL native interface database connection is open

Import Data from MySQL Database

`sqlouterjoin` Outer join between two MySQL database tables
`sqlinnerjoin` Inner join between two MySQL database tables
`sqlfind` Find information about all table types in MySQL database
`sqlread` Import data into MATLAB from MySQL database table
`fetch` Import results of SQL statement in MySQL database into MATLAB
`executeSQLScript` Execute SQL script on MySQL database

Export Data to MySQL Database

`sqlwrite` Insert MATLAB data into MySQL database table

MySQL Database Operations

`execute` Execute SQL statement using MySQL native interface database connection
`commit` Make changes to MySQL database permanent
`rollback` Undo changes to MySQL database
`sqlupdate` Update rows in MySQL database table

Examples

Connect to MySQL Database Using MySQL Native Interface

Create a MySQL® native interface connection to a MySQL database. Then, import data from the database into MATLAB® and perform simple data analysis. Close the database connection.

This example assumes that you are connecting to a MySQL database using the MariaDB® C Connector driver.

Connect to the database using the data source name, user name, and password.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";

conn = mysql(datasource,username,password)

conn =
    connection with properties:
        DataSource: "MySQLNative"
```

```

        UserName: "root"

Database Properties:
        AutoCommit: "on"
        LoginTimeout: 0
        MaxDatabaseConnections: 0

Catalog and Schema Information:
        DefaultCatalog: "toy_store"
        Catalogs: ["information_schema", "mysql", "performance_schema" ... and 3 more]
        Schemas: []

Database and Driver Information:
        DatabaseProductName: "MySQL"
        DatabaseProductVersion: "8.0.3-rc-log"
        DriverName: "Mariadb Connector/C"
        DriverVersion: "3.2.5"

```

The property sections of the `connection` object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `sqlread` function. Display the first three rows of data.

```

tablename = "inventoryTable";
data = sqlread(conn,tablename);
head(data,3)

```

productNumber	Quantity	Price	inventoryDate
1	1700	15	23-Sep-2014 13:38:34
2	1200	9	09-Jul-2014 02:50:45
3	356	17	14-May-2014 11:14:28

Determine the highest product quantity from the table.

```
max(data.Quantity)
```

```
ans = 9000
```

Close the database connection `conn`.

```
close(conn)
```

Version History

Introduced in R2020b

See Also

`close` | `fetch` | `sqlread` | `sqlupdate`

Topics

“Import Data from MySQL Database Table” on page 6-6

SQLConnectionOptions

Define MySQL native interface database connection options

Description

Create connection options for a MySQL native interface connection.

First, create an `SQLConnectionOptions` object, set the connection options, test the connection, and save the data source. Then, create a MySQL native interface connection using the saved data source. The connection options include the options required to make a database connection. You can also define additional connection options for a specific database driver.

Creation

Create an `SQLConnectionOptions` object using the `databaseConnectionOptions` function.

Properties

DataSourceName — Data source name

string scalar

Data source name, specified as a string scalar. You can use the data source name in the `mysql` function to create a database connection for the MySQL native interface.

Example: "MySQLDataSource"

Data Types: string

Vendor — Database vendor

string scalar

This property is read-only.

Database vendor, specified as a string scalar. Specify this property using the `vendor` input argument in the `databaseConnectionOptions` function. After the `SQLConnectionOptions` object exists, you cannot set this property to another value.

Example: "MySQL"

Data Types: string

DatabaseName — Database name

string scalar

Database name, specified as a string scalar. Set this property using the `setoptions` function.

Example: "toystore_doc"

Data Types: string

Server — Database server name or address

"localhost" (default) | string scalar

Database server name or address, specified as a string scalar. Set this property using the `setoptions` function.

Data Types: string

PortNumber — Server port number where the server is listening

numeric scalar

Server port number where the server is listening, specified as a numeric scalar. The default value is 3306 for a MySQL database. Set this property using the `setoptions` function.

Data Types: double

Object Functions

<code>rmoptions</code>	Remove MySQL native interface connection options
<code>saveAsDataSource</code>	Save MySQL native interface data source
<code>setoptions</code>	Set MySQL native interface connection options
<code>reset</code>	Reset MySQL native interface connection options to defaults
<code>testConnection</code>	Test MySQL native interface database connection

Examples**Create MySQL Native Interface Data Source**

Create, configure, test, and save a MySQL® native interface data source for a MySQL database.

Create a MySQL native interface data source for a MySQL native interface database connection.

```
vendor = "MySQL";
opts = databaseConnectionOptions("native", vendor)
```

```
opts =
    SQLConnectionOptions with properties:
```

```
        DataSourceName: ""
            Vendor: "MySQL"

        DatabaseName: ""
            Server: "localhost"
            PortNumber: 3306
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Configure the data source by setting the database connection options for the data source `MySQLDataSource`, database name `toystore_doc`, database server `dbtb01`, and port number `3306`.

```
opts = setoptions(opts, ...
    'DataSourceName','MySQLDataSource', ...
    'DatabaseName','toystore_doc','Server','dbtb01', ...
    'PortNumber',3306)
```

```
opts =
    SQLConnectionOptions with properties:

        DataSourceName: "MySQLDataSource"
            Vendor: "MySQL"

        DatabaseName: "toystore_doc"
            Server: "dbtb01"
            PortNumber: 3306
```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection with a user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "root";
password = "matlab";
status = testConnection(opts,username,password)

status = logical
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `mysql` function or the Database Explorer app.

Version History

Introduced in R2020b

See Also

`databaseConnectionOptions` | `deleteDataSource`

Topics

“Configure MySQL Native Interface Data Source” on page 6-2

“Import Data from MySQL Database Table” on page 6-6

mysql

Create MySQL native interface database connection

Syntax

```
conn = mysql(datasource,username,password)
conn = mysql(username,password,Name,Value)
```

Description

`conn = mysql(datasource,username,password)` creates a MySQL native interface database connection using the specified data source, user name, and password. `conn` is a connection object.

`conn = mysql(username,password,Name,Value)` creates a MySQL native interface database connection using the specified user name and password, with additional options specified by one or more name-value pair arguments. For example, "Server", "dbtb00" specifies the database server name as dbtb00.

Examples

Connect to MySQL Database Using MySQL Native Interface

Create a MySQL® native interface connection to a MySQL database. Then, import data from the database into MATLAB® and perform simple data analysis. Close the database connection.

This example assumes that you are connecting to a MySQL database using the MariaDB® C Connector driver.

Connect to the database using the data source name, user name, and password.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";

conn = mysql(datasource,username,password)

conn =
  connection with properties:

      DataSource: "MySQLNative"
      UserName: "root"

  Database Properties:

      AutoCommit: "on"
      LoginTimeout: 0
      MaxDatabaseConnections: 0

  Catalog and Schema Information:
```

```

DefaultCatalog: "toy_store"
Catalogs: ["information_schema", "mysql", "performance_schema" ... and 3 more
Schemas: []

```

Database and Driver Information:

```

DatabaseProductName: "MySQL"
DatabaseProductVersion: "8.0.3-rc-log"
DriverName: "Mariadb Connector/C"
DriverVersion: "3.2.5"

```

The property sections of the `connection` object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `sqlread` function. Display the first three rows of data.

```

tablename = "inventoryTable";
data = sqlread(conn,tablename);
head(data,3)

```

productNumber	Quantity	Price	inventoryDate
1	1700	15	23-Sep-2014 13:38:34
2	1200	9	09-Jul-2014 02:50:45
3	356	17	14-May-2014 11:14:28

Determine the highest product quantity from the table.

```
max(data.Quantity)
```

```
ans = 9000
```

Close the database connection `conn`.

```
close(conn)
```

Connect to MySQL Database Using MySQL Native Interface and Additional Options

Create a MySQL® native interface connection to a MySQL database using name-value pair arguments. Then, import data from the database into MATLAB® and perform simple data analysis. Close the database connection.

This example assumes that you are connecting to a MySQL database version 5.7.22 using the MySQL Connector/C++ driver version 8.0.15.

Connect to the database using the user name and password shown. Specify the database server name `dbtb01`, database name `toystore_doc`, and port number `3306` by setting the corresponding name-value pair arguments.

```

username = "root";
password = "matlab";

conn = mysql(username,password,'Server',"dbtb01", ...
    'DatabaseName',"toystore_doc",'PortNumber',3306)

conn =
    connection with properties:

        Database: "toystore_doc"
        UserName: "root"

Database Properties:

        AutoCommit: "on"
        LoginTimeout: 0
        MaxDatabaseConnections: 151

Catalog and Schema Information:

        DefaultCatalog: "toystore_doc"
        Catalogs: ["information_schema", "mysql", "performance_schema" ... and 3 more
        Schemas: []

Database and Driver Information:

        DatabaseProductName: "MySQL"
        DatabaseProductVersion: "5.7.22"
        DriverName: "MySQL Connector/C++"
        DriverVersion: "8.0.15"

```

The property sections of the connection object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `sqlread` function. Display the first three rows of data.

```

tablename = "inventoryTable";
data = sqlread(conn,tablename);
head(data,3)

```

```

ans=3x4 table
    productNumber    Quantity    Price    inventoryDate
    _____    _____    _____    _____
         1           1700         14.5    "2014-09-23 09:38:34"
         2           1200          9      "2014-07-08 22:50:45"
         3           356          17      "2014-05-14 07:14:28"

```

Determine the highest product quantity from the table.

```

max(data.Quantity)

```

```
ans = 9000
```

Close the database connection `conn`.

```
close(conn)
```

Input Arguments

datasource — Data source name

character vector | string scalar

Data source name, specified as a character vector or string scalar. Specify the name of an existing data source.

Example: "myDataSource"

Data Types: char | string

username — User name

character vector | string scalar

User name required to access the database, specified as a character vector or string scalar. If no user name is required, specify an empty value "".

Data Types: char | string

password — Password

character vector | string scalar

Password required to access the database, specified as a character vector or string scalar. If no password is required, specify an empty value "".

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `conn =`

```
mysql(username,password,"Server","dbtb01","PortNumber",3306,"DatabaseName","toystore_doc")
```

 creates a MySQL native interface database connection using the database server `dbtb01`, port number 3306, and database name `toystore_doc`.

Server — Database server name

"localhost" (default) | string scalar | character vector

Database server name or address, specified as the comma-separated pair consisting of 'Server' and a string scalar or character vector.

Example: "dbtb00"

Data Types: char | string

PortNumber — Port number

3306 (default) | numeric scalar

Port number, specified as the comma-separated pair consisting of 'PortNumber' and a numeric scalar.

Example: 3306

Data Types: double

DatabaseName — Database name

"" (default) | string scalar | character vector

Database name, specified as the comma-separated pair consisting of 'DatabaseName' and a string scalar or character vector. If you do not specify a database name, the `mysql` function connects to the default database on the database server.

Example: "toystore_doc"

Data Types: char | string

Version History

Introduced in R2020b

See Also

`databaseConnectionOptions` | `close` | `isopen` | `sqlread`

Topics

"Configure MySQL Native Interface Data Source" on page 6-2

"Import Data from MySQL Database Table" on page 6-6

close

Package: database.mysql

Close MySQL native interface database connection

Syntax

```
close(conn)
```

Description

`close(conn)` closes and invalidates the MySQL native interface database connection to free up database and driver resources.

Examples

Connect to MySQL Database Using MySQL Native Interface

Create a MySQL® native interface connection to a MySQL database. Then, import data from the database into MATLAB® and perform simple data analysis. Close the database connection.

This example assumes that you are connecting to a MySQL database using the MariaDB® C Connector driver.

Connect to the database using the data source name, user name, and password.

```
datasource = "MySQLNative";  
username = "root";  
password = "matlab";
```

```
conn = mysql(datasource,username,password)
```

```
conn =
```

```
connection with properties:
```

```
DataSource: "MySQLNative"  
UserName: "root"
```

```
Database Properties:
```

```
AutoCommit: "on"  
LoginTimeout: 0  
MaxDatabaseConnections: 0
```

```
Catalog and Schema Information:
```

```
DefaultCatalog: "toy_store"  
Catalogs: ["information_schema", "mysql", "performance_schema" ... and 3 more  
Schemas: []
```

```
Database and Driver Information:
```



```

DatabaseProductName: "MySQL"
DatabaseProductVersion: "8.0.3-rc-log"
DriverName: "Mariadb Connector/C"
DriverVersion: "3.2.5"

```

The property sections of the `connection` object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `sqlread` function. Display the first three rows of data.

```

tablename = "inventoryTable";
data = sqlread(conn,tablename);
head(data,3)

```

<u>productNumber</u>	<u>Quantity</u>	<u>Price</u>	<u>inventoryDate</u>
1	1700	15	23-Sep-2014 13:38:34
2	1200	9	09-Jul-2014 02:50:45
3	356	17	14-May-2014 11:14:28

Determine the highest product quantity from the table.

```
max(data.Quantity)
```

```
ans = 9000
```

Close the database connection `conn`.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

Version History

Introduced in R2020b

See Also

`mysql` | `fetch`

Topics

“Configure MySQL Native Interface Data Source” on page 6-2

“Import Data from MySQL Database Table” on page 6-6

commit

Package: database.mysql

Make changes to MySQL database permanent

Syntax

```
commit(conn)
```

Description

`commit(conn)` makes changes to the database connection `conn` permanent, specifically any changes made since the last `commit` or `rollback` function has been run. To use the `commit` function, you must set the `AutoCommit` property of the connection object to `off`.

Examples

Commit Data to MySQL Database

Use a MySQL® native interface database connection to insert product data from MATLAB® into a new table in a MySQL database. Then, commit the changes to the database.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Allow manual committing of changes to the database by setting the `AutoCommit` property to `off`.

```
conn.AutoCommit = "off";
```

Create a MATLAB table that contains data for two products. The data is stored in the `productTable` and `suppliers` tables.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"],'VariableNames',["productNumber" ...
    "stockNumber" "supplierNumber" "unitCost" "productDescription"]);
```

Insert the product data into a new table named `toyTable`.

```
tablename = "toyTable";
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
-----
           30      5e+05      1000           25      "Rubik's Cube"
           40      6e+05      2000           30      "Doll House"
```

Commit the changes to the database.

```
commit(conn)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

Version History

Introduced in R2020b

See Also

`mysql` | `execute` | `sqlwrite` | `rollback`

Topics

“Roll Back Data in Database Using MySQL Native Interface” on page 6-16

“Create Table and Add Column Using MySQL Native Interface” on page 6-18

“Delete Data from Database Using MySQL Native Interface” on page 6-19

reset

Package: `database.options.native.mysql`

Reset MySQL native interface connection options to defaults

Syntax

```
opts = reset(opts)
```

Description

`opts = reset(opts)` resets all connection options to their default values for all properties of the `SQLConnectionOptions` object. The `reset` function also removes any additional driver-specific properties from the `SQLConnectionOptions` object.

Examples

Reset Connection Options to Default Values

Create, configure, and test a MySQL® native interface data source for a MySQL database. Reset the database connection options to their default values. Then configure, test, and save the data source with different database connection options.

Create a MySQL native interface data source for a MySQL database connection.

```
vendor = "MySQL";  
opts = databaseConnectionOptions("native", vendor)
```

```
opts =  
  SQLConnectionOptions with properties:  
    DataSourceName: ""  
    Vendor: "MySQL"  
  
    DatabaseName: ""  
    Server: "localhost"  
    PortNumber: 3306
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Configure the data source by setting the database connection options for the data source `MySQLDataSource`, database name `toystore_doc`, database server `dbtb01`, and port number `3306`.

```
opts = setoptions(opts, ...
    'DataSourceName','MySQLDataSource', ...
    'DatabaseName','toystore_doc','Server','dbtb01', ...
    'PortNumber',3306)
```

```
opts =
    SQLConnectionOptions with properties:

        DataSourceName: "MySQLDataSource"
        Vendor: "MySQL"

        DatabaseName: "toystore_doc"
        Server: "dbtb01"
        PortNumber: 3306
```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection with a user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "root";
password = "matlab";
status = testConnection(opts,username,password)

status = logical
        1
```

Reset the database connection options to their default values.

```
opts = reset(opts)

opts =
    SQLConnectionOptions with properties:

        DataSourceName: ""
        Vendor: "MySQL"

        DatabaseName: ""
        Server: "localhost"
        PortNumber: 3306
```

Configure the data source again by setting the database connection options for the data source `MySQLDataSource`, database name `toystore_doc`, database server `dbtb01`, and port number `3306`. Also, set a driver-specific connection option to specify a timeout value for establishing the database connection.

```
opts = setoptions(opts, ...
    "DataSourceName","MySQLDataSource", ...
    "DatabaseName","toystore_doc", ...
    "Server","dbtb01","PortNumber",3306, ...
    "OPT_CONNECT_TIMEOUT",20)
```

```

opts =
  SqlConnectionOptions with properties:

        DataSourceName: "MySQLDataSource"
        Vendor: "MySQL"

        DatabaseName: "toystore_doc"
        Server: "dbtb01"
        PortNumber: 3306

  Additional Connection Options:

        OPT_CONNECT_TIMEOUT: 20

```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SqlConnectionOptions` object. The driver-specific connection option appears below the other connection options.

Test the database connection again.

```

username = "root";
password = "matlab";
status = testConnection(opts,username,password)

status = logical
      1

```

Save the configured data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

`SqlConnectionOptions` object

Database connection options, specified as an `SqlConnectionOptions` object.

Output Arguments

opts — Database connection options

`SqlConnectionOptions` object

Database connection options, returned as an `SqlConnectionOptions` object.

Version History

Introduced in R2020b

See Also

Objects

SQLConnectionOptions

Functions

databaseConnectionOptions | roptions | saveAsDataSource | setoptions |
testConnection | deleteDataSource | mysql

Topics

“Configure MySQL Native Interface Data Source” on page 6-2

“Import Data from MySQL Database Table” on page 6-6

“Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface”
on page 6-8

rmoptions

Package: database.options.native.mysql

Remove MySQL native interface connection options

Syntax

```
opts = rmoptions(opts,option)
```

Description

`opts = rmoptions(opts,option)` removes one or more specified connection options from the `SQLConnectionOptions` object `opts`.

Examples

Remove Driver-Specific Connection Option

Edit an existing MySQL® native interface data source for a MySQL database. Set an additional driver-specific option, and test the database connection. Then, remove the additional driver-specific option, and test and save the data source.

Retrieve the existing MySQL native interface data source.

```
datasource = "MySQLDataSource";
opts = databaseConnectionOptions(datasource)

opts =
  SQLConnectionOptions with properties:

      DataSourceName: "MySQLDataSource"
      Vendor: "MySQL"

      DatabaseName: "toystore_doc"
      Server: "dbtb01"
      PortNumber: 3306
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Add a driver-specific connection option by using a name-value pair argument. The option specifies a timeout value for establishing the database connection. `opts` contains a new property for the additional connection option.

```
opts = setoptions(opts,"OPT_CONNECT_TIMEOUT",20)
```

```
opts =  
  SQLConnectionOptions with properties:  
      DataSourceName: "MySQLDataSource"  
      Vendor: "MySQL"  
  
      DatabaseName: "toystore_doc"  
      Server: "dbtb01"  
      PortNumber: 3306  
  
  Additional Connection Options:  
      OPT_CONNECT_TIMEOUT: 20
```

Test the database connection with a user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "root";  
password = "matlab";  
status = testConnection(opts,username,password)  
  
status = logical  
      1
```

Remove the driver-specific option for specifying a timeout value. The `opts` object no longer contains the `OPT_CONNECT_TIMEOUT` property.

```
opts = rmoptions(opts,"OPT_CONNECT_TIMEOUT")  
  
opts =  
  SQLConnectionOptions with properties:  
      DataSourceName: "MySQLDataSource"  
      Vendor: "MySQL"  
  
      DatabaseName: "toystore_doc"  
      Server: "dbtb01"  
      PortNumber: 3306
```

Test the database connection again.

```
username = "root";  
password = "matlab";  
status = testConnection(opts,username,password)  
  
status = logical  
      1
```

Save the data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, specified as an SQLConnectionOptions object.

option — MySQL native interface connection option

character vector | string scalar | cell array of character vectors | string array

MySQL native interface connection option, specified as a character vector, string scalar, cell array of character vectors, or string array. Specify the name of one or more MySQL native interface connection options or driver-specific connection options.

Example: ["sslCert" "OPT_RECONNECT"]

Example: "OPT_CONNECT_TIMEOUT"

Data Types: char | string | cell

Output Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, returned as an SQLConnectionOptions object.

Version History

Introduced in R2020b

See Also

Objects

SQLConnectionOptions

Functions

databaseConnectionOptions | mysql | reset | setoptions | testConnection | deleteDataSource | saveAsDataSource

Topics

“Configure MySQL Native Interface Data Source” on page 6-2

“Import Data from MySQL Database Table” on page 6-6

“Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface” on page 6-8

saveAsDataSource

Package: database.options.native.mysql

Save MySQL native interface data source

Syntax

```
saveAsDataSource(opts)
```

Description

`saveAsDataSource(opts)` saves the MySQL native interface data source specified by the `SQLConnectionOptions` object `opts`.

Examples

Create MySQL Native Interface Data Source

Create, configure, test, and save a MySQL® native interface data source for a MySQL database.

Create a MySQL native interface data source for a MySQL native interface database connection.

```
vendor = "MySQL";  
opts = databaseConnectionOptions("native", vendor)
```

```
opts =  
  SQLConnectionOptions with properties:
```

```
      DataSourceName: ""  
      Vendor: "MySQL"  
  
      DatabaseName: ""  
      Server: "localhost"  
      PortNumber: 3306
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Configure the data source by setting the database connection options for the data source `MySQLDataSource`, database name `toystore_doc`, database server `dbtb01`, and port number `3306`.

```
opts = setoptions(opts, ...  
  'DataSourceName', "MySQLDataSource", ...
```

```

    'DatabaseName', "toystore_doc", 'Server', "dbtb01", ...
    'PortNumber', 3306)

opts =
    SQLConnectionOptions with properties:

        DataSourceName: "MySQLDataSource"
        Vendor: "MySQL"

        DatabaseName: "toystore_doc"
        Server: "dbtb01"
        PortNumber: 3306

```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection with a user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```

username = "root";
password = "matlab";
status = testConnection(opts,username,password)

status = logical
        1

```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `mysql` function or the Database Explorer app.

Input Arguments

opts — Database connection options

`SQLConnectionOptions` object

Database connection options, specified as an `SQLConnectionOptions` object.

Version History

Introduced in R2020b

See Also

Objects

`SQLConnectionOptions`

Functions

`databaseConnectionOptions` | `mysql` | `reset` | `roptions` | `setoptions` | `testConnection` | `deleteDataSource`

Topics

“Configure MySQL Native Interface Data Source” on page 6-2

“Import Data from MySQL Database Table” on page 6-6

“Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface”
on page 6-8

setoptions

Package: database.options.native.mysql

Set MySQL native interface connection options

Syntax

```
opts = setoptions(opts,Option1,OptionValue1,...,OptionN,OptionValueN)
```

Description

`opts = setoptions(opts,Option1,OptionValue1,...,OptionN,OptionValueN)` sets connection options using the `SQLConnectionOptions` object `opts`.

Examples

Create MySQL Native Interface Data Source

Create, configure, test, and save a MySQL® native interface data source for a MySQL database.

Create a MySQL native interface data source for a MySQL native interface database connection.

```
vendor = "MySQL";
opts = databaseConnectionOptions("native", vendor)
```

```
opts =
  SQLConnectionOptions with properties:
```

```
      DataSourceName: ""
      Vendor: "MySQL"

      DatabaseName: ""
      Server: "localhost"
      PortNumber: 3306
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Configure the data source by setting the database connection options for the data source `MySQLDataSource`, database name `toystore_doc`, database server `dbtb01`, and port number `3306`.

```
opts = setoptions(opts, ...
  'DataSourceName', "MySQLDataSource", ...
```

```

        'DatabaseName', "toystore_doc", 'Server', "dbtb01", ...
        'PortNumber', 3306)

opts =
    SQLConnectionOptions with properties:

        DataSourceName: "MySQLDataSource"
        Vendor: "MySQL"

        DatabaseName: "toystore_doc"
        Server: "dbtb01"
        PortNumber: 3306

```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection with a user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```

username = "root";
password = "matlab";
status = testConnection(opts,username,password)

status = logical
         1

```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `mysql` function or the Database Explorer app.

Edit Existing MySQL Native Interface Data Source

Edit an existing MySQL® native interface data source for a MySQL database. Set an additional driver-specific option and save the data source.

Retrieve the existing MySQL data source `MySQLDataSource`.

```

datasource = "MySQLDataSource";
opts = databaseConnectionOptions(datasource)

opts =
    SQLConnectionOptions with properties:

        DataSourceName: "MySQLDataSource"
        Vendor: "MySQL"

        DatabaseName: "toystore_doc"
        Server: "dbtb01"
        PortNumber: 3306

```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Add a driver-specific connection option by using a name-value pair argument. The option specifies a timeout value for establishing the database connection. `opts` contains a new section of properties for the additional connection option.

```
opts = setoptions(opts,"OPT_CONNECT_TIMEOUT",20)
```

```
opts =
  SqlConnectionOptions with properties:

      DataSourceName: "MySQLDataSource"
      Vendor: "MySQL"

      DatabaseName: "toystore_doc"
      Server: "dbtb01"
      PortNumber: 3306

  Additional Connection Options:

      OPT_CONNECT_TIMEOUT: 20
```

Test the database connection with a user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "root";
password = "matlab";
status = testConnection(opts,username,password)

status = logical
      1
```

Save the updated data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

`SqlConnectionOptions` object

Database connection options, specified as an `SqlConnectionOptions` object.

Option1,OptionValue1,...,OptionN,OptionValueN — MySQL native interface connection options

name-value pair arguments

MySQL native interface connection options, specified as one or more name-value pair arguments. `Option` is a character vector or string scalar that specifies the name of a MySQL native interface

connection option. `OptionValue` specifies the value of the option. `OptionValue` can be a character vector, string scalar, logical scalar, or numeric scalar. You can specify any MySQL native interface connection option that is a property of the `SQLConnectionOptions` object. You can also set driver-specific connection options.

Example: `'DataSourceName', "myDataSource", 'Server', "localhost", 'PortNumber', 3306` configures a MySQL native interface data source named `myDataSource` that is located on the local server with the port number 3306.

Output Arguments

opts — Database connection options

`SQLConnectionOptions` object

Database connection options, returned as an `SQLConnectionOptions` object.

Version History

Introduced in R2020b

See Also

Objects

`SQLConnectionOptions`

Functions

`databaseConnectionOptions` | `mysql` | `reset` | `roptions` | `testConnection` | `deleteDataSource` | `saveAsDataSource`

Topics

“Configure MySQL Native Interface Data Source” on page 6-2

“Import Data from MySQL Database Table” on page 6-6

“Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface” on page 6-8

testConnection

Package: database.options.native.mysql

Test MySQL native interface database connection

Syntax

```
status = testConnection(opts,username,password)
[status,message] = testConnection(opts,username,password)
```

Description

`status = testConnection(opts,username,password)` tests the MySQL native interface database connection specified by the `SQLConnectionOptions` object `opts`, a user name, and a password.

`[status,message] = testConnection(opts,username,password)` also returns the error message associated with testing the database connection.

Examples

Create MySQL Native Interface Data Source

Create, configure, test, and save a MySQL® native interface data source for a MySQL database.

Create a MySQL native interface data source for a MySQL native interface database connection.

```
vendor = "MySQL";
opts = databaseConnectionOptions("native",vendor)
```

```
opts =
  SQLConnectionOptions with properties:
      DataSourceName: ""
      Vendor: "MySQL"
      DatabaseName: ""
      Server: "localhost"
      PortNumber: 3306
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Configure the data source by setting the database connection options for the data source `MySQLDataSource`, database name `toystore_doc`, database server `dbtb01`, and port number `3306`.

```
opts = setoptions(opts, ...  
    'DataSourceName','MySQLDataSource', ...  
    'DatabaseName','toystore_doc','Server','dbtb01', ...  
    'PortNumber',3306)
```

```
opts =  
    SQLConnectionOptions with properties:  
  
        DataSourceName: "MySQLDataSource"  
        Vendor: "MySQL"  
  
        DatabaseName: "toystore_doc"  
        Server: "dbtb01"  
        PortNumber: 3306
```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection with a user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "root";  
password = "matlab";  
status = testConnection(opts,username,password)  
  
status = logical  
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `mysql` function or the Database Explorer app.

Retrieve Message for MySQL Native Interface Database Connection Test

Create and configure a MySQL native interface data source to a MySQL database. Test the database connection to the MySQL native interface data source and retrieve the error message.

Create a MySQL native interface data source for a MySQL database connection.

```
vendor = "MySQL";  
opts = databaseConnectionOptions("native",vendor)  
  
opts =  
    SQLConnectionOptions with properties:  
  
        DataSourceName: ""  
        Vendor: "MySQL"
```

```

DatabaseName: ""
Server: "localhost"
PortNumber: 3306

```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Configure the data source by setting the database connection options for the data source `MySQLDataSource`, database name `toystore_doc`, database server `dbtb01`, and port number `3306`.

```

opts = setoptions(opts, ...
    'DataSourceName', "MySQLDataSource", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb01", ...
    'PortNumber', 3306)

```

`opts =`
`SQLConnectionOptions` with properties:

```

DataSourceName: "MySQLDataSource"
Vendor: "MySQL"

DatabaseName: "toystore_doc"
Server: "dbtb01"
PortNumber: 3306

```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection using an incorrect user name and password. The `testConnection` function returns the logical `0`, which indicates the database connection fails. Retrieve and display the error message for the failed connection.

```

username = "wronguser";
password = "wrongpassword";
[status,message] = testConnection(opts,username,password)

status = logical
    0

message =
'1045 : Access denied for user 'wronguser'@'SERVERNAME' (using password: YES).'
```

Input Arguments

opts — Database connection options
`SQLConnectionOptions` object

Database connection options, specified as an `SQLConnectionOptions` object.

username — User name

character vector | string scalar

User name required to access the database, specified as a character vector or string scalar. If no user name is required, specify an empty value `''`.

Data Types: `char` | `string`

password — Password

character vector | string scalar

Password required to access the database, specified as a character vector or string scalar. If no password is required, specify an empty value `''`.

Data Types: `char` | `string`

Output Arguments

status — Connection status

logical

Connection status, returned as a logical `true` if the connection test passes or `false` if the connection test fails.

message — Error message

character vector

Error message, returned as a character vector. If the connection test passes, then the error message is an empty character vector. Otherwise, the error message contains text describing the failed database connection.

Version History

Introduced in R2020b

See Also

Objects

`SQLConnectionOptions`

Functions

`databaseConnectionOptions` | `mysql` | `reset` | `roptions` | `setoptions` | `deleteDataSource` | `saveAsDataSource`

Topics

“Configure MySQL Native Interface Data Source” on page 6-2

“Import Data from MySQL Database Table” on page 6-6

“Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface” on page 6-8

execute

Package: database.mysql

Execute SQL statement using MySQL native interface database connection

Syntax

```
execute(conn,sqlquery)
```

Description

`execute(conn,sqlquery)` executes an SQL query that contains a non-SELECT SQL statement by using the relational database connection.

Examples

Execute Non-SELECT SQL Statement

Using a relational database connection, create and execute a non-SELECT SQL statement that deletes a database table. The `MySQLNative` data source configures a database connection to a MySQL® database.

This example uses a MySQL database version 5.7.22 database and the MySQL Connector/C++ driver version 8.0.15.

Connect to the database using the data source name, user name, and password.

```
datasource = "MySQLNative";  
username = "root";  
password = "matlab";
```

```
conn = mysql(datasource,username,password);
```

Load patient information into the MATLAB® workspace.

```
patients = readtable("patients.xls");
```

Create the `patients` database table using the patient information.

```
tablename = "patients";  
sqlwrite(conn,tablename,patients)
```

Import the data from the `patients` database table.

```
data = sqlread(conn,tablename);
```

Delete the `patients` database table using the `execute` function.

```
sqlquery = strcat("DROP TABLE ",tablename);  
execute(conn,sqlquery)
```

Ensure that the table no longer exists.

```
data = sqlfind(conn,tablename)
```

```
data =
```

```
    0x5 empty table
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a `connection` object.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar. The SQL statement can be any valid non-SELECT SQL statement.

The SQL statement can be a stored procedure that does not return any result sets. For stored procedures that return one or more result sets, use the `fetch` function.

For information about the SQL query language, see the SQL Tutorial.

Example: "DROP TABLE patients"

Data Types: char | string

Version History

Introduced in R2020b

See Also

`mysql` | `close` | `fetch` | `sqlread`

Topics

"Create Table and Add Column Using MySQL Native Interface" on page 6-18

"Delete Data from Database Using MySQL Native Interface" on page 6-19

External Websites

MySQL Documentation

executeSQLScript

Package: database.mysql

Execute SQL script on MySQL database

Syntax

```
results = executeSQLScript(conn,scriptfile)
results = executeSQLScript(conn,scriptfile,Name,Value)
```

Description

`results = executeSQLScript(conn,scriptfile)` uses the database connection `conn` to return a structure array that contains results as a table (by default) for each executed SQL SELECT statement in the SQL script file. For any non-SELECT SQL statements, the corresponding table is empty. The `executeSQLScript` function executes all SQL statements in the SQL script file.

`results = executeSQLScript(conn,scriptfile,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'DataReturnFormat','cellarray'` stores the results of an executed SQL statement as a cell array. The results are stored in the `Data` field of the `results` structure array.

Examples

Execute SQL Script Using MySQL Native Interface

Connect to a MySQL® database. Then, run two SQL SELECT statements from the SQL script file `compare_sales.sql`, import the results, and perform simple sales data analysis. The file contains two SQL queries: the first retrieves sales of products from US suppliers, and the second retrieves sales of products from foreign suppliers.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Execute the SQL script. The SQL script contains two SQL queries that retrieve sales data from US and foreign suppliers, respectively.

```
scriptfile = "compare_sales.sql";
results = executeSQLScript(conn,scriptfile)
```

```
results=1x2 struct array with fields:
    SQLQuery
    Data
    Message
```

The `executeSQLScript` function returns a structure array that contains two tables in the `Data` field. The first table contains the results of executing the first SQL query in the SQL script file. The second table contains the results of executing the second SQL query.

Display the first eight rows of imported data for the second SQL query in the SQL script file. The data shows sales results from foreign suppliers.

```
data = head(results(2).Data)
```

```
data=4x6 table
  productDescription      supplierName      city      Jan_Sales      Feb_Sales      Mar_Sales
  _____      _____      _____      _____      _____      _____
  "Victorian Doll"      "Wacky Widgets"      "Adelaide"      1400      1100      9800
  "Painting Set"      "Terrific Toys"      "London"      3000      2400      1800
  "Sail Boat"      "Incredible Machines"      "Dublin"      3000      2400      1500
  "Slinky"      "Doll's Galore"      "London"      3000      1500      1000
```

Retrieve the variable names in the table.

```
names = data.Properties.VariableNames
```

```
names = 1x6 cell
  {'productDescription'}      {'supplierName'}      {'city'}      {'Jan_Sales'}      {'Feb_Sales'}      {'Mar_Sales'}
```

Determine the highest sales amount in January.

```
max(data.Jan_Sales)
```

```
ans = 3000
```

Close the database connection.

```
close(conn)
```

Execute SQL Script and Return Results as Structures

Connect to a MySQL® database. Then, run two SQL `SELECT` statements from the SQL script file `compare_sales.sql`. Import the results from the SQL queries as structures and perform simple sales data analysis. The file contains two SQL queries: the first SQL query retrieves sales of products from US suppliers, and the second retrieves sales of products from foreign suppliers.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Execute the SQL script. The SQL script contains two SQL queries that retrieve sales data from US and foreign suppliers, respectively. Specify `structure` as the data return format for the query results.

```

scriptfile = "compare_sales.sql";
results = executeSQLScript(conn,scriptfile, ...
    'DataReturnFormat','structure')

results=1x2 struct array with fields:
    SQLQuery
    Data
    Message

```

The `executeSQLScript` function returns a structure array that contains two structures in the `Data` field. The first structure contains the results of executing the first SQL query in the SQL script file. The second structure contains the results of executing the second SQL query.

Display the imported data for the second SQL query in the SQL script file. The data contains sales results from foreign suppliers.

```

data = results(2).Data

data = struct with fields:
    productDescription: [4x1 string]
    supplierName: [4x1 string]
    city: [4x1 string]
    Jan_Sales: [4x1 double]
    Feb_Sales: [4x1 double]
    Mar_Sales: [4x1 double]

```

Determine the highest sales amount in January.

```

max(data.Jan_Sales)

ans = 3000

```

Close the database connection.

```

close(conn)

```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

scriptfile — Name of SQL script file

character vector | string scalar

Name of SQL script file that contains one or more SQL statements to run, specified as a character vector or string scalar. The file must be a text file and can contain comments in addition to SQL queries. Start single-line comments with `--`. Enclose multiline comments in `/*...*/`.

The SQL script file can contain one or more SQL statements terminated by either a semicolon or the keyword `GO`. The following is an example of two SQL `SELECT` statements.

```

SELECT productDescription, supplierName
FROM suppliers A, productTable B

```

```
WHERE A.SupplierNumber = B.SupplierNumber;
```

```
SELECT supplierName, Country  
FROM suppliers;
```

Example: 'C:\work\sql_file.sql'

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . ,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `results = executeSQLScript(conn,scriptfile,'DataReturnFormat','numeric','ErrorHandling','store')` returns query results as a numeric matrix in the `Data` field of the `results` structure array and stores any error message in the `Message` field of `results`.

DataReturnFormat — Data return format

'table' (default) | 'cellarray' | 'numeric' | 'structure'

Data return format, specified as the comma-separated pair consisting of 'DataReturnFormat' and one of these values:

- 'table'
- 'cellarray'
- 'numeric'
- 'structure'

You can specify the value using a character vector or string scalar.

The 'DataReturnFormat' name-value pair argument specifies the data type of the `Data` field in the `results` structure array.

Example: 'DataReturnFormat','structure' returns a structure array that contains query results stored in structures.

ErrorHandling — Error handling

'report' (default) | 'store'

Error handling, specified as the comma-separated pair consisting of 'ErrorHandling' and one of these values:

- 'report' — When an SQL statement fails to execute, stop execution of the remaining SQL statements in the SQL script file and display an error message at the command line.
- 'store' — When an SQL statement fails to execute, store an error message in the `Message` field of the `results` structure array.

You can specify the value using a character vector or string scalar.

Example: 'ErrorHandling','report' displays an error message at the command line.

Output Arguments

results — Query results

structure array

Query results from executed SQL statements in the SQL script file, returned as a structure array with these fields.

Field Name	Field Data Type	Field Description
SQLQuery	character vector	Stores the SQL statement or statements executed in the SQL script file.
Data	<ul style="list-style-type: none"> • table (default) • cell array • numeric matrix • structure 	<p>Stores the results of executed SQL SELECT statements.</p> <p>The 'DataReturnFormat' name-value pair argument specifies the data type of the Data field.</p> <p>For non-SELECT SQL statements, the Data field is an empty double, which means the executed SQL query has no results.</p>
Message	character vector	<p>Stores an error message for the respective SQL statement that fails to execute.</p> <p>The Message field contains an error message only if you specify the 'ErrorHandling' name-value pair argument with the value 'store'.</p>

The number of elements in the structure array is equal to the number of SQL statements in the SQL script file. `results(M)` contains the results from executing the Mth SQL statement in the SQL script file. If the SQL statement returns query results, then the results are stored in `results(M).Data`.

For details about accessing structure arrays, see "Structure Arrays".

Limitations

- Use the `executeSQLScript` function to import data into MATLAB, especially if you have long and complex SQL queries that are difficult to convert into MATLAB character vectors or string scalars. The `executeSQLScript` function does not support SQL scripts containing continuous PL/SQL blocks with `BEGIN` and `END`, such as stored procedure definitions or trigger definitions. However, `executeSQLScript` does support table definitions.
- An SQL script containing either of the following can produce unexpected results:
 - Apostrophes that are not escaped, including those in comments. For example, write the character vector 'Here's the code' as 'Here''s the code'.
 - Nested comments.
- An SQL script containing more than 25,000 characters causes the `executeSQLScript` function to return an error.

Version History

Introduced in R2020b

See Also

`mysql` | `close` | `rollback` | `commit`

Topics

“Import Data from MySQL Database Table” on page 6-6

“Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface” on page 6-8

“Import Large Data Using DatabaseDatastore Object and MySQL Native Interface” on page 6-11

“Roll Back Data in Database Using MySQL Native Interface” on page 6-16

External Websites

MySQL Documentation

fetch

Package: `database.mysql`

Import results of SQL statement in MySQL database into MATLAB

Syntax

```
results = fetch(conn,sqlquery)
results = fetch(conn,sqlquery,opts)
results = fetch( ___,Name,Value)
[results,metadata] = fetch( ___ )
```

Description

`results = fetch(conn,sqlquery)` returns all rows of data after executing the SQL statement `sqlquery` for the connection object. `fetch` imports data in batches.

`results = fetch(conn,sqlquery,opts)` customizes options for importing data from an executed SQL query by using the `SQLImportOptions` object.

`results = fetch(___,Name,Value)` specifies additional options using one or more name-value arguments with any of the previous input argument combinations. For example, specify `MaxRows = 5` to import five rows of data.

`[results,metadata] = fetch(___)` also returns the `metadata` table, which contains metadata information about the imported data.

Examples

Import All Data Using MySQL Native Interface

Import all product data from a MySQL® database table into MATLAB® using the MySQL native interface and the `fetch` function. Determine the highest unit cost among products in the table. Then, use a row filter to import only the data for products with a unit cost less than 15.

Create a MySQL native interface database connection to a MySQL database using a data source, username, and password. The database contains the table `productTable`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Import all the data from `productTable` by using the connection object and SQL query. Then, display the first five rows of the imported data.

```
sqlquery = "SELECT * FROM productTable";
data = fetch(conn,sqlquery);
head(data,5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	1.2597e+05	1003	13	"Victorian Doll"
8	2.1257e+05	1001	5	"Train Set"
7	3.8912e+05	1007	16	"Engine Kit"
2	4.0031e+05	1002	9	"Painting Set"
4	4.0034e+05	1008	21	"Space Cruiser"

Determine the highest unit cost for all products in the table.

```
max(data.unitCost)
```

```
ans = 24
```

Now, import the data using a row filter. The filter condition is that unitCost must be less than 15.

```
rf = rowfilter("unitCost");
rf = rf.unitCost < 15;
data = fetch(conn,sqlquery,"RowFilter",rf);
```

Again, display the first five rows of the imported data.

```
head(data,5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	1.2597e+05	1003	13	"Victorian Doll"
8	2.1257e+05	1001	5	"Train Set"
2	4.0031e+05	1002	9	"Painting Set"
1	4.0034e+05	1001	14	"Building Blocks"
5	4.0046e+05	1005	3	"Tin Soldier"

Close the database connection.

```
close(conn)
```

Import Data from SQL Query Using Import Options

Customize import options when importing data from the results of an SQL query on a MySQL® database using the MySQL native interface. Control the import options by creating an `SQLImportOptions` object. Then, customize import options for different columns in the SQL query. Import data using the `fetch` function.

This example uses the `employees_database.mat` file, which contains the columns `first_name`, `hire_date`, and `DEPARTMENT_NAME`. The example assumes that you are connecting to a MySQL database version 5.7.22 using the MySQL Connector/C++ driver version 8.0.15.

Create a MySQL native interface database connection to a MySQL database with a data source name, username, and password.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
```



```
conn = mysql(datasource,username,password);
```

Load employee information into the MATLAB® workspace.

```
employeeedata = load("employees_database.mat");
```

Create the employees and departments database tables using the employee information.

```
emps = employeeedata.employees;
depts = employeeedata.departments;
```

```
sqlwrite(conn,"employees",emps)
sqlwrite(conn,"departments",depts)
```

Create an SQLImportOptions object using an SQL query and the databaseImportOptions function. This query retrieves all information for employees who are sales managers or programmers.

```
% sqlquery = strcat("SELECT * from employees e join departments d ", ...
%     "on (e.department_id = d.department_id) WHERE ", ...
%     "(job_id = 'IT_PROG' or job_id = 'SA_MAN')");
sqlquery = ['SELECT first_name, last_name, hire_date, salary, DEPARTMENT_NAME ' ...
'from employees e join departments d ' ...
'on (e.department_id = d.department_id) where job_id ' ...
'in ('IT_PROG','SA_MAN')'];
opts = databaseImportOptions(conn,sqlquery)
```

```
opts =
  SQLImportOptions with properties:
```

```
    ExcludeDuplicates: false
    VariableNamingRule: 'preserve'

    VariableNames: {'first_name', 'last_name', 'hire_date' ... and 2 more}
    VariableTypes: {'string', 'string', 'datetime' ... and 2 more}
    SelectedVariableNames: {'first_name', 'last_name', 'hire_date' ... and 2 more}
    FillValues: { <missing>, <missing>, NaT ... and 2 more }
    RowFilter: <unconstrained>

    VariableOptions: Show all 5 VariableOptions
```

Display the current import options for the variables selected in the SelectedVariableNames property of the SQLImportOptions object.

```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)
```

```
varOpts =
  1x5 SQLVariableImportOptions array with properties:
```

```
Variable Options:
      (1) |      (2) |      (3) |      (4) |      (5)
Name: 'first_name' | 'last_name' | 'hire_date' | 'salary' | 'DEPARTMENT_NAME'
Type: 'string' | 'string' | 'datetime' | 'double' | 'string'
MissingRule: 'fill' | 'fill' | 'fill' | 'fill' | 'fill'
FillValue: <missing> | <missing> | NaT | NaN | <missing>
```

To access sub-properties of each variable, use getoptions

Change the data types for the `hire_date`, `DEPARTMENT_NAME`, and `first_name` variables using the `setoptions` function. Then, display the updated import options. For efficiency, change the data type of the `hire_date` variable to `string`. Because `DEPARTMENT_NAME` designates a finite set of repeating values, change the data type of this variable to `categorical`. Also, change the name of this variable to lowercase. Because `first_name` stores text data, change the data type of this variable to `char`.

```
opts = setoptions(opts,"hire_date","Type","string");
opts = setoptions(opts,"DEPARTMENT_NAME","Name","department_name", ...
    "Type","categorical");
opts = setoptions(opts,"first_name","Type","char");
```

```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)
```

```
varOpts =
    1x5 SQLVariableImportOptions array with properties:
```

Variable Options:

	(1)	(2)	(3)	(4)	(5)
Name:	'first_name'	'last_name'	'hire_date'	'salary'	'department_name'
Type:	'char'	'string'	'string'	'double'	'categorical'
MissingRule:	'fill'	'fill'	'fill'	'fill'	'fill'
FillValue:	''	<missing>	<missing>	NaN	<undefined>

To access sub-properties of each variable, use `getoptions`

Select the three modified variables using the `SelectVariableNames` property.

```
opts.SelectedVariableNames = ["first_name","hire_date","department_name"];
```

Set the filter condition to import only the data for the employees hired before January 1, 2006.

```
opts.RowFilter = opts.RowFilter.hire_date < datetime(2006,01,01)
```

```
opts =
    SQLImportOptions with properties:
```

```
    ExcludeDuplicates: false
    VariableNamingRule: 'preserve'

    VariableNames: {'first_name', 'last_name', 'hire_date' ... and 2 more}
    VariableTypes: {'char', 'string', 'string' ... and 2 more}
    SelectedVariableNames: {'first_name', 'hire_date', 'department_name'}
    FillValues: {'', <missing>, <missing> ... and 2 more }
    RowFilter: hire_date < 01-Jan-2006
```

VariableOptions: Show all 5 VariableOptions

Import and display the results of the SQL query using the `fetch` function.

```
employees_data = fetch(conn,sqlquery,opts)
```

```
employees_data=4x3 table
    first_name      hire_date      department_name
    _____      _____      _____
```

```

{'David' }    "25-Jun-2005"    IT
{'John'  }    "01-Oct-2004"    Sales
{'Karen' }    "05-Jan-2005"    Sales
{'Alberto'}  "10-Mar-2005"    Sales

```

Delete the `employees` and `departments` database tables using the `execute` function.

```

execute(conn,"DROP TABLE employees")
execute(conn,"DROP TABLE departments")

```

Close the database connection.

```
close(conn)
```

Import Data from SQL Query as Structure

Specify the data return format and the number of imported rows for the results of an SQL query. Import data using the SQL query and the `fetch` function.

This example assumes that you are connecting to a MySQL® database version 5.7.22 using the MySQL Connector/C++ driver version 8.0.15.

Create a MySQL native interface database connection to a MySQL database with a data source name, user name, and password.

```

datasource = "MySQLNative";
username = "root";
password = "matlab";

```

```
conn = mysql(datasource,username,password);
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the `patients` database table using the patient information.

```
tablename = 'patients';
sqlwrite(conn,tablename,patients)
```

Select all data from the `patients` database table and import five rows from the table as a structure. Use the `'DataReturnFormat'` name-value pair argument to specify returning the data as a structure. Also, use the `'MaxRows'` name-value pair argument to specify five rows. Display the imported data.

```

sqlquery = strcat("SELECT * FROM ",tablename);
results = fetch(conn,sqlquery,'DataReturnFormat','structure', ...
    'MaxRows',5)

```

```

results = struct with fields:
    LastName: [5×1 string]
    Gender: [5×1 string]
    Age: [5×1 double]
    Location: [5×1 string]

```

```
        Height: [5×1 double]
        Weight: [5×1 double]
        Smoker: [5×1 logical]
        Systolic: [5×1 double]
        Diastolic: [5×1 double]
SelfAssessedHealthStatus: [5×1 string]
```

Delete the `patients` database table using the `execute` function.

```
sqlquery = strcat("DROP TABLE ",tablename);
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Retrieve Metadata Information About Imported Data

Retrieve metadata information when importing data from an SQL query. Import data using the `fetch` function and explore the metadata information by using dot notation.

This example uses the `outages.csv` file, which contains outage data. Also, the example assumes that you are connecting to a MySQL® database version 5.7.22 using the MySQL Connector/C++ driver version 8.0.15.

Create a MySQL native interface database connection to a MySQL database with a data source name, user name, and password.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
```

```
conn = mysql(datasource,username,password);
```

Load outage information into the MATLAB® workspace.

```
outages = readtable("outages.csv");
```

Create the `outages` database table using the outage information. Use the `'ColumnType'` name-value pair argument to customize the data types of the variables in the `outages` table.

```
tablename = "outages";
sqlwrite(conn,tablename,outages, ...
    'ColumnType',[ "varchar(120)", "datetime", "numeric(38,16)", ...
    "numeric(38,16)", "datetime", "varchar(150)"])
```

Import the data into the MATLAB workspace and return metadata information about the imported data.

```
sqlquery = "SELECT * FROM outages";
[results,metadata] = fetch(conn,sqlquery);
```

View the names of the variables in the imported data.

```
metadata.Properties.RowNames
```

```
ans = 6x1 cell
    {'Region'      }
    {'OutageTime'  }
    {'Loss'        }
    {'Customers'   }
    {'RestorationTime'}
    {'Cause'       }
```

View the data type of each variable in the imported data.

`metadata.VariableType`

```
ans = 6x1 cell
    {'string' }
    {'datetime'}
    {'double' }
    {'double' }
    {'datetime'}
    {'string' }
```

View the missing data value for each variable in the imported data.

`metadata.FillValue`

```
ans=6x1 cell array
    {1x1 missing}
    {[NaT      ]}
    {[      NaN]}
    {[      NaN]}
    {[NaT      ]}
    {1x1 missing}
```

View the indices of the missing data for each variable in the imported data.

`metadata.MissingRows`

```
ans=6x1 cell array
    { 0x1 double}
    { 0x1 double}
    {1208x1 double}
    { 656x1 double}
    { 58x1 double}
    { 0x1 double}
```

Display the first eight rows of the imported data that contain missing restoration time values. `data` contains restoration time values in the fifth variable. Use the numeric indices to find the rows with missing data.

```
index = metadata.MissingRows{5,1};
nullrestoration = results(index,:);
head(nullrestoration)
```

```
ans=8x6 table
    Region      OutageTime      Loss      Customers      RestorationTime      Cause
```

"SouthEast"	23-Jan-2003 00:49:00	530.14	2.1204e+05	NaN	"winter st
"NorthEast"	18-Sep-2004 05:54:00	0	0	NaN	"equipment
"MidWest"	20-Apr-2002 16:46:00	23141	NaN	NaN	"unknown"
"NorthEast"	16-Sep-2004 19:42:00	4718	NaN	NaN	"unknown"
"SouthEast"	14-Sep-2005 15:45:00	1839.2	3.4144e+05	NaN	"severe st
"SouthEast"	17-Aug-2004 17:34:00	624.1	1.7879e+05	NaN	"severe st
"SouthEast"	28-Jan-2006 23:13:00	498.78	NaN	NaN	"energy em
"West"	20-Jun-2003 18:22:00	0	0	NaN	"energy em

Delete the `outages` database table using the `execute` function.

```
sqlstr = "DROP TABLE ";
sqlquery = strcat(sqlstr,tablename);
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar. The SQL statement can be any valid SQL statement, including nested queries. The SQL statement can be a stored procedure, such as `{call sp_name (parm1,parm2,...)}`. For stored procedures that return one or more result sets, use the `fetch` function.

Data Types: `char` | `string`

opts — Database import options

SQLImportOptions object

Database import options, specified as an `SQLImportOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `results = fetch(conn,sqlquery, 'MaxRows',50, 'DataReturnFormat', 'structure')` imports 50 rows of data as a structure.

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of 'MaxRows' and a positive numeric scalar. By default, the `fetch` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: 'MaxRows', 10

Data Types: double

DataReturnFormat — Data return format

'table' (default) | 'cellarray' | 'numeric' | 'structure'

Data return format, specified as the comma-separated pair consisting of 'DataReturnFormat' and one of these values:

- 'table'
- 'cellarray'
- 'numeric'
- 'structure'

Use the 'DataReturnFormat' name-value pair argument to specify the data type of the results data. To specify integer classes for numeric data, use the `opts` input argument.

You can specify the value using a character vector or string scalar.

Example: 'DataReturnFormat', 'cellarray' imports data as a cell array.

VariableNamingRule — Variable naming rule

"preserve" (default) | "modify"

Variable naming rule, specified as the comma-separated pair consisting of 'VariableNamingRule' and one of these values:

- "preserve" — Preserve most variable names when the `fetch` function imports data. For details, see the Limitations section.
- "modify" — Remove non-ASCII characters from variable names when the `fetch` function imports data.

Example: 'VariableNamingRule', "modify"

Data Types: string

RowFilter — Row filter condition

<unconstrained> (default) | matlab.io.RowFilter object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; fetch(conn,sqlquery,"RowFilter",rf)`

Output Arguments**results — Result data**

table (default) | cell array | structure | numeric matrix

Result data, returned as a table, cell array, structure, or numeric matrix. The result data contains all rows of data from the executed SQL statement by default.

Use the 'MaxRows' name-value pair argument to specify the number of rows of data to import. Use the 'DataReturnFormat' name-value pair argument to specify the data type of the result data.

When the executed SQL statement does not return any rows, the result data is an empty table.

When you import data, the fetch function converts the data type of each column from the MySQL database to the MATLAB data type. This table maps the data type of a database column to the converted MATLAB data type.

MySQL Data Type	MATLAB Data Type
BIT	logical
TINYINT	double
SMALLINT	double
BIGINT	double
REAL	double
DOUBLE	double
DECIMAL	double
NUMERIC	double
CHAR	string
VARCHAR	string
LONGVARCHAR	string
TIMESTAMP	datetime
DATE	datetime
TIME	duration
YEAR	double
ENUM	categorical
JSON	char

metadata – Metadata information

table

Metadata information, returned as a table with these variables.

Variable Name	Variable Description	Variable Data Type
VariableType	Data type of each variable in the imported data	Cell array of character vectors
FillValue	Value of missing data for each variable in the imported data	Cell array of missing data values
MissingRows	Indices for each occurrence of missing data in each variable of the imported data	Cell array of numeric indices

By default, the `fetch` function imports text data as a character vector and numeric data as a double. `FillValue` is an empty character array (for text data) or `NaN` (for numeric data) by default. To change the missing data value to another value, use the `SQLImportOptions` object.

The `RowNames` property of the metadata table contains the names of the variables in the imported data.

Limitations

The name-value argument `VariableNamingRule` has these limitations:

- The `fetch` function returns an error if you specify the `VariableNamingRule` name-value argument and set the `DataReturnFormat` name-value argument to `"cellarray"`, `"structure"`, or `"numeric"`.
- The `fetch` function returns a warning if you set the `VariableNamingRule` property of the `SQLImportOptions` object to `"preserve"` and set the `DataReturnFormat` name-value argument to `"structure"`.
- The `fetch` function returns an error if you use the `VariableNamingRule` name-value argument with the `SQLImportOptions` object `opts`.
- If you set the `VariableNamingRule` name-value argument to the value `"modify"`:
 - These variable names are reserved identifiers for the table data type: `Properties`, `RowNames`, and `VariableNames`.
 - The length of each variable name must be less than the number returned by `namelengthmax`.

The name-value argument `RowFilter` has this limitation:

- The `fetch` function returns an error if you specify the `RowFilter` name-value argument with the `SQLImportOptions` object `opts`. It is ambiguous which of the `RowFilter` object to use in this case, especially if the filter conditions are different.

Alternative Functionality

App

The `fetch` function imports data using the command line. To import data interactively, use the **Database Explorer** app.

Version History

Introduced in R2020b

R2023a: Selectively import rows of data based on filter condition

You can use the `RowFilter` name-value argument to selectively import rows of data from a database table.

See Also

Functions

`close` | `mysql` | `databaseImportOptions` | `getoptions` | `reset` | `setoptions` | `execute`

Topics

“Import Data from MySQL Database Table” on page 6-6

“Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface” on page 6-8

“Import Large Data Using DatabaseDatastore Object and MySQL Native Interface” on page 6-11

“Delete Data from Database Using MySQL Native Interface” on page 6-19

External Websites

MySQL Documentation

isopen

Package: database.mysql

Determine if MySQL native interface database connection is open

Syntax

```
i = isopen(conn)
```

Description

`i = isopen(conn)` returns 1 if the MySQL native interface database connection is open and 0 if it is closed or invalid.

Examples

Determine If Database Connection Is Open

Connect to a MySQL® database using the MySQL native interface and verify the database connection. Then, import data from the database into MATLAB® using the database table `productTable`. Determine the highest unit cost among the retrieved products in the table. Close the database connection and ensure that the connection is closed.

Create a MySQL native interface database connection using a data source, user name, and password. The MySQL database contains the table `productTable`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Determine if the database connection is open. The `isopen` function returns the logical 1, which means the database connection is open.

```
i = isopen(conn)
```

```
i = logical
     1
```

Select all the data from `productTable` and sort it by the product number. `data` is a table containing the imported data that results from executing the SQL `SELECT` statement.

```
sqlquery = "SELECT * FROM productTable ORDER BY productNumber";
data = fetch(conn,sqlquery);
```

Display the first three rows of data.

```
head(data,3)
```

```
ans=3x5 table
    productNumber    stockNumber    supplierNumber    unitCost    productDescription
```

1	4.0035e+05	1001	14	"Building Blocks"
2	4.0031e+05	1002	9	"Painting Set"
3	4.01e+05	1009	17	"Slinky"

Determine the highest unit cost in the table.

```
max(data.unitCost)
```

```
ans = 24
```

Close the database connection.

```
close(conn)
```

Determine if the database connection is closed. The `isopen` function returns the logical `0`, which means the database connection is closed. If the database connection is invalid, the `isopen` function returns the same result.

```
i = isopen(conn)
```

```
i = logical  
0
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

Version History

Introduced in R2020b

See Also

`mysql` | `close` | `fetch`

Topics

“Configure MySQL Native Interface Data Source” on page 6-2

“Import Data from MySQL Database Table” on page 6-6

rollback

Package: database.mysql

Undo changes to MySQL database

Syntax

```
rollback(conn)
```

Description

`rollback(conn)` reverses changes made to a database using functions such as `sqlwrite`. The `rollback` function reverses all changes made since the last COMMIT or ROLLBACK operation. To use this function, you must set the `AutoCommit` property of the connection object to `off`.

Examples

Reverse Changes Made to MySQL Database

Use a MySQL® native interface database connection to insert product data from MATLAB® into a new table in a MySQL database. Then, reverse the changes made to the database.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Allow manual committing of changes to the database by setting the `AutoCommit` property to `off`.

```
conn.AutoCommit = "off";
```

Create a MATLAB table that contains data for two products. The data is stored in the `productTable` and `suppliers` tables.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"],'VariableNames',["productNumber" ...
    "stockNumber" "supplierNumber" "unitCost" "productDescription"]);
```

Insert the product data into a new table named `toyTable`.

```
tablename = "toyTable";
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
-----
           30      5e+05      1000           25      "Rubik's Cube"
           40      6e+05      2000           30      "Doll House"
```

Reverse the changes made to the database.

```
rollback(conn)
```

Import and display the contents of the database table again. The results are empty.

```
rows = sqlread(conn,tablename)
```

```
rows =
```

```
  0x5 empty table
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

Version History

Introduced in R2020b

See Also

`mysql` | `commit` | `sqlwrite` | `execute`

Topics

“Roll Back Data in Database Using MySQL Native Interface” on page 6-16

“Create Table and Add Column Using MySQL Native Interface” on page 6-18

“Delete Data from Database Using MySQL Native Interface” on page 6-19

sqlfind

Package: database.mysql

Find information about all table types in MySQL database

Syntax

```
data = sqlfind(conn,pattern)
data = sqlfind(conn,pattern,'Catalog',catalog)
```

Description

`data = sqlfind(conn,pattern)` returns information about all the “Table Types” on page 11-550 in a database where the specified character pattern appears in the name of the table type. Executing this function is the equivalent of writing the SQL statement `SELECT * FROM information_schema.tables`.

`data = sqlfind(conn,pattern,'Catalog',catalog)` finds table types in the specified catalog.

Examples

Find Information About Table Types Using MySQL Native Interface

Use a MySQL® native interface database connection to find information about all database table types in a MySQL database.

Create a MySQL native interface database connection to a MySQL database.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";

conn = mysql(datasource,username,password);
```

Find information about all table types in the database.

```
data = sqlfind(conn,"");
```

Display information about the first three table types.

```
head(data,3)
```

```
ans=3x5 table
```

Catalog	Schema	Table	Columns	Type
"mysql"	"	"columns_priv"	{1×7 string}	"TABLE"
"mysql"	"	"db"	{1×22 string}	"TABLE"
"mysql"	"	"engine_cost"	{1×6 string}	"TABLE"

`data` contains these variables:

- Catalog name
- Schema name
- Table name
- Columns in the table type
- Table type

Close the database connection.

```
close(conn)
```

Find Information About Table Types in Catalog

Use a MySQL® native interface database connection to find information about all database table types in a MySQL database. Specify the database catalog to search.

Create a MySQL native interface database connection to a MySQL database.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Find information about all table types in the `mysql` database catalog. Use the 'Catalog' name-value pair argument to specify the catalog. `data` is a table that contains information about all the table types in the specified catalog.

```
data = sqlfind(conn,"",'Catalog','mysql');
```

Display the first eight table types.

```
head(data)
```

```
ans=8x5 table
```

Catalog	Schema	Table	Columns	Type
"mysql"	"	"columns_priv"	{1x7 string}	"TABLE"
"mysql"	"	"db"	{1x22 string}	"TABLE"
"mysql"	"	"engine_cost"	{1x6 string}	"TABLE"
"mysql"	"	"event"	{1x22 string}	"TABLE"
"mysql"	"	"func"	{1x4 string}	"TABLE"
"mysql"	"	"general_log"	{1x6 string}	"TABLE"
"mysql"	"	"gtid_executed"	{1x3 string}	"TABLE"
"mysql"	"	"help_category"	{1x4 string}	"TABLE"

`data` contains these variables:

- Catalog name
- Schema name

- Table name
- Columns in the database table
- Table type

Display the column names in the fourth table type.

```
data.Columns{4}
```

```
ans = 1x22 string
      "db"      "name"      "body"      "definer"      "execute_at"      "interval_value"      "interval_field"
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

pattern — Pattern

character vector | string scalar

Pattern, specified as a character vector or string scalar. The `sqlfind` function searches for this text in the names of the table types in a database. To find all table types, specify an empty character vector or string scalar.

Example: "inventory"

Data Types: char | string

catalog — Database catalog name

character vector | string scalar

Database catalog name, specified as a character vector or string scalar. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have numerous catalogs.

Example: 'toy_store'

Data Types: char | string

Output Arguments

data — Table type information

table

Table type information, returned as a table that contains information about the table types in the database, where the table type name partially or fully matches the text in `pattern`. The data output contains these variables in a string array.

Variable	Description
Catalog	Catalog name where the database table type is stored
Schema	Schema name where the database table type is stored
Table	Database table name
Columns	Column names in the database table type
Type	Database table type

More About

Table Types

Table types are a subset of database objects that store or reference data.

The `sqlfind` function recognizes these table types in a database:

- Table
- View
- System table
- System view
- Synonym
- Global temporary table
- Local temporary table

Version History

Introduced in R2020b

See Also

`sqlread` | `mysql` | `close` | `sqlinnerjoin`

Topics

“Import Data from MySQL Database Table” on page 6-6

“Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface” on page 6-8

“Import Large Data Using DatabaseDatastore Object and MySQL Native Interface” on page 6-11

“Create Table and Add Column Using MySQL Native Interface” on page 6-18

“Delete Data from Database Using MySQL Native Interface” on page 6-19

sqlinnerjoin

Package: database.mysql

Inner join between two MySQL database tables

Syntax

```
data = sqlinnerjoin(conn, lefttable, righttable)
data = sqlinnerjoin(conn, lefttable, righttable, Name, Value)
```

Description

`data = sqlinnerjoin(conn, lefttable, righttable)` returns a table resulting from an inner join between the left and right database tables. This function matches rows using all shared columns, or keys, in both database tables. The inner join retains only the rows that match between the two tables. Executing this function is the equivalent of writing the SQL statement `SELECT * FROM lefttable, righttable INNER JOIN lefttable.key = righttable.key`.

`data = sqlinnerjoin(conn, lefttable, righttable, Name, Value)` uses additional options specified by one or more name-value arguments. For example, specify `Keys = "productNumber"` to use the `productNumber` column as a key for joining the two database tables.

Examples

Join Two Database Tables Using MySQL Native Interface

Use a MySQL® native interface database connection to import product data from an inner join between two MySQL database tables into MATLAB®.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource, username, password);
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlinnerjoin` function automatically detects the shared column between the tables. `data` is a table that contains the matched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlinnerjoin(conn, lefttable, righttable);
```

Display the first three rows of matched data. The columns from the right table (`suppliers`) appear to the right of the columns from the left table (`productTable`).

```
head(data, 3)
```

```
ans=3x10 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription  Supplier
  _____  _____  _____  _____  _____  _____
           8      2.1257e+05      1001           5      "Train Set"          100
           1      4.0035e+05      1001          14      "Building Blocks"    100
           2      4.0031e+05      1002           9      "Painting Set"       100
```

Close the database connection.

```
close(conn)
```

Specify Key for Joining Two Database Tables

Use a MySQL® native interface database connection to import joined product data from two MySQL database tables into MATLAB®. Specify the key to use for joining the tables.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. Specify the key, or shared column, between the tables using the 'Keys' name-value pair argument. `data` is a table that contains the matched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlinnerjoin(conn,lefttable,righttable,'Keys',"supplierNumber");
```

Display the first three rows of matched data. The columns from the right table (`suppliers`) appear to the right of the columns from the left table (`productTable`).

```
head(data,3)
```

```
ans=3x10 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription  Supplier
  _____  _____  _____  _____  _____  _____
           8      2.1257e+05      1001           5      "Train Set"          100
           1      4.0035e+05      1001          14      "Building Blocks"    100
           2      4.0031e+05      1002           9      "Painting Set"       100
```

Close the database connection.

```
close(conn)
```

Filter Rows in Joined Data

Use a MySQL® native interface database connection to import joined product data from two MySQL database tables into MATLAB®. Specify the row filter condition to use for joining the tables.

Create a MySQL native interface database connection to a MySQL database using the data source name, username, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Join the two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlinnerjoin` function automatically detects the shared column between the tables. The table `data` contains the matched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlinnerjoin(conn,lefttable,righttable);
```

Display the first five rows of matched data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0034e+05	1001	14	"Building Blocks"	1001
1	4.0034e+05	1001	14	"Building Blocks"	1001
2	4.0031e+05	1002	9	"Painting Set"	1002
2	4.0031e+05	1002	9	"Painting Set"	1002
3	4.01e+05	1009	17	"Slinky"	1009

Join the same tables, but this time use a row filter. The filter condition is that `unitCost` must be less than 10. Again, display the first five rows of matched data.

```
rf = rowfilter("unitCost");
rf = rf.unitCost <= 10;
data = sqlinnerjoin(conn,lefttable,righttable,"RowFilter",rf);
head(data,5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
2	4.0031e+05	1002	9	"Painting Set"	1002
2	4.0031e+05	1002	9	"Painting Set"	1002
5	4.0046e+05	1005	3	"Tin Soldier"	1005
5	4.0046e+05	1005	3	"Tin Soldier"	1005
6	4.0088e+05	1004	8	"Sail Boat"	1004

Close the database connection.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

lefttable — Left table

character vector | string scalar

Left table, specified as a character vector or string scalar. Specify the name of the database table on the left side of the join.

Example: 'inventoryTable'

Data Types: char | string

righttable — Right table

character vector | string scalar

Right table, specified as a character vector or string scalar. Specify the name of the database table on the right side of the join.

Example: 'productTable'

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `data =`

```
sqlinnerjoin(conn, "productTable", "suppliers", 'LeftCatalog', 'toy_store', 'RightCatalog', 'toy_shop', 'MaxRows', 5)
```

 performs an inner join between left and right tables by specifying the catalog for both tables and returns five matched rows.

LeftCatalog — Left catalog

character vector | string scalar

Left catalog, specified as the comma-separated pair consisting of 'LeftCatalog' and a character vector or string scalar. Specify the database catalog name where the left table of the join is stored.

Example: 'LeftCatalog', 'toy_store'

Data Types: char | string

RightCatalog — Right catalog

character vector | string scalar

Right catalog, specified as the comma-separated pair consisting of 'RightCatalog' and a character vector or string scalar. Specify the database catalog name where the right table of the join is stored.

Example: 'RightCatalog', 'toy_store'

Data Types: char | string

Keys — Keys

character vector | string scalar | cell array of character vectors | string array

Keys, specified as the comma-separated pair consisting of 'Keys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. Use this name-value pair argument to identify the shared keys (columns) between the two tables to join.

You cannot use this name-value pair argument with the 'LeftKeys' and 'RightKeys' name-value pair arguments.

Example: 'Keys', 'MANAGER_ID'

Data Types: char | string | cell

LeftKeys — Left keys

character vector | string scalar | cell array of character vectors | string array

Left keys, specified as the comma-separated pair consisting of 'LeftKeys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the left table for the join to the right table.

Use this name-value pair argument with the 'RightKeys' name-value pair argument. Both arguments must specify the same number of keys. The `sqlinnerjoin` function pairs the values of the keys based on their order.

Example: 'LeftKeys', ["productNumber" "Price"], 'RightKeys', ["productNumber" "Price"]

Data Types: char | string | cell

RightKeys — Right keys

character vector | string scalar | cell array of character vectors | string array

Right keys, specified as the comma-separated pair consisting of 'RightKeys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the right table for the join to the left table.

Use this name-value pair argument with the 'LeftKeys' name-value pair argument. Both arguments must specify the same number of keys. The `sqlinnerjoin` function pairs the values of the keys based on their order.

Example: 'LeftKeys', ["productIdentifier" "Cost"], 'RightKeys', ["productNumber" "Price"]

Data Types: char | string | cell

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of 'MaxRows' and a positive numeric scalar. By default, the `sqlinnerjoin` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: 'MaxRows',10

Data Types: double

VariableNamingRule — Variable naming rule

"preserve" (default) | "modify"

Variable naming rule, specified as the comma-separated pair consisting of 'VariableNamingRule' and one of these values:

- "preserve" — Preserve most variable names when the `sqlinnerjoin` function imports data. For details, see the Limitations section.
- "modify" — Remove non-ASCII characters from variable names when the `sqlinnerjoin` function imports data.

Example: 'VariableNamingRule','modify'

Data Types: string

RowFilter — Row filter condition

<unconstrained> (default) | `matlab.io.RowFilter` object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; sqlinnerjoin(conn,lefttable,righttable,"RowFilter",rf)`

Output Arguments

data — Joined data

table

Joined data, returned as a table that contains the matched rows from the join of the left and right tables. `data` also contains a variable for each column in the left and right tables.

When you import data, the `sqlinnerjoin` function converts the data type of each column from the MySQL database to the MATLAB data type. This table maps the data type of a database column to the converted MATLAB data type.

MySQL Data Type	MATLAB Data Type
BIT	logical
TINYINT	double
SMALLINT	double
BIGINT	double
REAL	double
DOUBLE	double
DECIMAL	double
NUMERIC	double
CHAR	string
VARCHAR	string

MySQL Data Type	MATLAB Data Type
LONGVARCHAR	string
TIMESTAMP	datetime
DATE	datetime
TIME	duration
YEAR	double
ENUM	categorical
JSON	char

If the column names are shared between the joined database tables and have the same case, then the `sqlinnerjoin` function adds a unique suffix to the corresponding variable names in `data`.

Limitations

The name-value argument `VariableNamingRule` has these limitations if it is set to the value `"modify"`.

- The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the table data type.
- The length of each variable name must be less than the number returned by `namelengthmax`.

Version History

Introduced in R2020b

R2023a: Selectively join data based on filter condition

You can use the `RowFilter` when joining data from database tables.

See Also

`sqlread` | `sqlfind` | `sqlouterjoin` | `mysql` | `close`

Topics

"Import Data from MySQL Database Table" on page 6-6

sqlouterjoin

Package: database.mysql

Outer join between two MySQL database tables

Syntax

```
data = sqlouterjoin(conn, lefttable, righttable, 'Type', type)
data = sqlouterjoin(conn, lefttable, righttable, Name, Value)
```

Description

`data = sqlouterjoin(conn, lefttable, righttable, 'Type', type)` returns a table resulting from an outer join between the left and right database tables. Specify the join type to be a left or right join. This function matches rows using all shared columns, or keys, in both database tables. The outer join retains the matched and unmatched rows between the two tables. Executing this function is the equivalent of writing the SQL statement `SELECT * FROM lefttable LEFT JOIN righttable ON lefttable.key = righttable.key` or `SELECT * FROM lefttable RIGHT JOIN righttable ON lefttable.key = righttable.key`.

`data = sqlouterjoin(conn, lefttable, righttable, Name, Value)` uses additional options specified by one or more name-value arguments. For example, specify `Keys = "productNumber"` to use the `productNumber` column as a key for joining the two database tables.

Examples

Join Two Database Tables Using MySQL Native Interface

Use a MySQL® native interface database connection to import product data from an outer join between two MySQL database tables into MATLAB®.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource, username, password);
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlouterjoin` function automatically detects the shared column between the tables. Use the `'Type'` name-value pair argument to retrieve records that have matching values in the selected column of both tables, and unmatched records from the left table only. `data` is a table that contains the matched and unmatched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlouterjoin(conn, lefttable, righttable, 'Type', "left");
```

Display the first three rows of joined data. The columns from the right table (suppliers) appear to the right of the columns from the left table (productTable).

```
head(data,3)
```

```
ans=3x10 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription  Supplier
  _____  _____  _____  _____  _____  _____
           8      2.1257e+05      1001           5      "Train Set"          100
           1      4.0035e+05      1001          14      "Building Blocks"    100
           2      4.0031e+05      1002           9      "Painting Set"       100
```

Close the database connection.

```
close(conn)
```

Specify Key for Joining Two Database Tables

Use a MySQL® native interface database connection to import product data from an outer join between two MySQL database tables into MATLAB®.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the tables productTable and suppliers.

```
datasource = "MySQLDataSource";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Join two database tables, productTable and suppliers. The productTable table is the left table of the join, and the suppliers table is the right table of the join. The sqlouterjoin function automatically detects the shared column between the tables. Use the 'Type' name-value pair argument to retrieve records that have matching values in the selected column of both tables, and unmatched records from the left table only. Specify the key, or shared column, between the tables using the 'Keys' name-value pair argument. data is a table that contains the matched and unmatched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlouterjoin(conn,lefttable,righttable,'Type',"left",'Keys',"supplierNumber");
```

Display the first three rows of matched data. The columns from the right table (suppliers) appear to the right of the columns from the left table (productTable).

```
head(data,3)
```

```
ans=3x10 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription  Supplier
  _____  _____  _____  _____  _____  _____
           8      2.1257e+05      1001           5      "Train Set"          100
           1      4.0035e+05      1001          14      "Building Blocks"    100
```

2	4.0031e+05	1002	9	"Painting Set"	1002
---	------------	------	---	----------------	------

Close the database connection.

```
close(conn)
```

Filter Rows in Joined Data

Use a MySQL® native interface database connection to import product data from an outer join between two MySQL database tables into MATLAB®. Specify the row filter condition to use for joining the tables.

Create a MySQL native interface database connection to a MySQL database using the data source name, username, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "MySQLDataSource";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Join the two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlouterjoin` function automatically detects the shared column between the tables. Use `Type` to retrieve records that have matching values in the selected column of both tables, and unmatched records from the left table only. The table `data` contains the matched and unmatched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlouterjoin(conn,lefttable,righttable,"Type","left");
```

Display the first five rows of matched data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
1	4.0034e+05	1001	14	"Building Blocks"	1001
1	4.0034e+05	1001	14	"Building Blocks"	1001
2	4.0031e+05	1002	9	"Painting Set"	1002
2	4.0031e+05	1002	9	"Painting Set"	1002
3	4.01e+05	1009	17	"Slinky"	1009

Join the same tables, but this time use a row filter. The filter condition is that `unitCost` must be less than 10. Again, display the first five rows of matched data.

```
rf = rowfilter("unitCost");
rf = rf.unitCost <= 10;
data = sqlouterjoin(conn,lefttable,righttable, ...
    "Type","left", ...
    "RowFilter",rf);
head(data,5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription	Supplier
2	4.0031e+05	1002	9	"Painting Set"	1002
2	4.0031e+05	1002	9	"Painting Set"	1002
5	4.0046e+05	1005	3	"Tin Soldier"	1005
5	4.0046e+05	1005	3	"Tin Soldier"	1005
6	4.0088e+05	1004	8	"Sail Boat"	1004

Close the database connection.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

lefttable — Left table

character vector | string scalar

Left table, specified as a character vector or string scalar. Specify the name of the database table on the left side of the join.

Example: 'inventoryTable'

Data Types: char | string

righttable — Right table

character vector | string scalar

Right table, specified as a character vector or string scalar. Specify the name of the database table on the right side of the join.

Example: 'productTable'

Data Types: char | string

type — Outer join type

'left' | 'right'

Outer join type, specified as the comma-separated pair consisting of 'Type' and one of these values:

- 'left' — A left join retrieves records that have matching values in the selected column of both tables, and unmatched records from the left table only.
- 'right' — A right join retrieves records that have matching values in the selected column of both tables, and unmatched records from the right table only.

You can specify this value as a character vector or string scalar.

Example: 'Type', 'left'

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `data = sqlouterjoin(conn,"productTable","suppliers",'Type','left','MaxRows',5)` performs an outer left join between left and right tables and returns five rows of the joined data.

LeftCatalog — Left catalog

character vector | string scalar

Left catalog, specified as the comma-separated pair consisting of `'LeftCatalog'` and a character vector or string scalar. Specify the database catalog name where the left table of the join is stored.

Example: `'LeftCatalog','toy_store'`

Data Types: `char` | `string`

RightCatalog — Right catalog

character vector | string scalar

Right catalog, specified as the comma-separated pair consisting of `'RightCatalog'` and a character vector or string scalar. Specify the database catalog name where the right table of the join is stored.

Example: `'RightCatalog','toy_store'`

Data Types: `char` | `string`

Keys — Keys

character vector | string scalar | cell array of character vectors | string array

Keys, specified as the comma-separated pair consisting of `'Keys'` and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. Use this name-value pair argument to identify the shared keys (columns) between the two tables to join.

You cannot use this name-value pair argument with the `'LeftKeys'` and `'RightKeys'` name-value pair arguments.

Example: `'Keys','MANAGER_ID'`

Data Types: `char` | `string` | `cell`

LeftKeys — Left keys

character vector | string scalar | cell array of character vectors | string array

Left keys, specified as the comma-separated pair consisting of `'LeftKeys'` and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the left table for the join to the right table.

Use this name-value pair argument with the `'RightKeys'` name-value pair argument. Both arguments must specify the same number of keys. The `sqlouterjoin` function pairs the values of the keys based on their order.

Example: 'LeftKeys', ["productNumber" "Price"], 'RightKeys', ["productNumber" "Price"]

Data Types: char | string | cell

RightKeys — Right keys

character vector | string scalar | cell array of character vectors | string array

Right keys, specified as the comma-separated pair consisting of 'RightKeys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the right table for the join to the left table.

Use this name-value pair argument with the 'LeftKeys' name-value pair argument. Both arguments must specify the same number of keys. The `sqlouterjoin` function pairs the values of the keys based on their order.

Example: 'LeftKeys', ["productIdentifier" "Cost"], 'RightKeys', ["productNumber" "Price"]

Data Types: char | string | cell

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of 'MaxRows' and a positive numeric scalar. By default, the `sqlouterjoin` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: 'MaxRows', 10

Data Types: double

VariableNamingRule — Variable naming rule

"preserve" (default) | "modify"

Variable naming rule, specified as the comma-separated pair consisting of 'VariableNamingRule' and one of these values:

- "preserve" — Preserve most variable names when the `sqlouterjoin` function imports data. For details, see the Limitations section.
- "modify" — Remove non-ASCII characters from variable names when the `sqlouterjoin` function imports data.

Example: 'VariableNamingRule', "modify"

Data Types: string

RowFilter — Row filter condition

<unconstrained> (default) | `matlab.io.RowFilter` object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; sqlouterjoin(conn, lefttable, righttable, "RowFilter", rf)`

Output Arguments

data — Joined data

table

Joined data, returned as a table that contains rows matched by keys in the left and right database tables and the retained unmatched rows. `data` also contains a variable for each column in the left and right tables.

When you import data, the `sqlouterjoin` function converts the data type of each column from the MySQL database to the MATLAB data type. This table maps the data type of a database column to the converted MATLAB data type.

MySQL Data Type	MATLAB Data Type
BIT	logical
TINYINT	double
SMALLINT	double
BIGINT	double
REAL	double
DOUBLE	double
DECIMAL	double
NUMERIC	double
CHAR	string
VARCHAR	string
LONGVARCHAR	string
TIMESTAMP	datetime
DATE	datetime
TIME	duration
YEAR	double
ENUM	categorical
JSON	char

If the column names are shared between the joined database tables and have the same case, then the `sqlouterjoin` function adds a unique suffix to the corresponding variable names in `data`.

The variables in `data` that correspond to columns in the left table contain NULL values when no matched rows exist in the right database table. Similarly, the variables that correspond to columns in the right table contain NULL values when no matched rows exist in the left database table.

Limitations

The name-value argument `VariableNamingRule` has these limitations if it is set to the value "modify".

- The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the `table` data type.

- The length of each variable name must be less than the number returned by `nameLengthmax`.

Version History

Introduced in R2020b

R2023a: Selectively join data based on filter condition

You can use the `RowFilter` when joining data from database tables.

See Also

`sqlfind` | `sqlinnerjoin` | `sqlread` | `mysql` | `close`

Topics

“Import Data from MySQL Database Table” on page 6-6

sqlread

Package: database.mysql

Import data into MATLAB from MySQL database table

Syntax

```
data = sqlread(conn,tablename)
data = sqlread(conn,tablename,opts)
data = sqlread( ___,Name,Value)
[data,metadata] = sqlread( ___)
```

Description

`data = sqlread(conn,tablename)` returns a table by importing data into MATLAB from a MySQL database table. Executing this function is the equivalent of writing a `SELECT * FROM tablename` SQL statement in ANSI SQL.

`data = sqlread(conn,tablename,opts)` customizes options for importing data from a database table using the `SQLImportOptions` object.

`data = sqlread(___,Name,Value)` specifies additional options using one or more name-value arguments with any of the previous input argument combinations. For example, specify `Catalog = "cat"` to import data from a database table stored in the "cat" catalog.

`[data,metadata] = sqlread(___)` also returns the metadata table, which contains metadata information about the imported data.

Examples

Import Data from Database Table Using MySQL Native Interface

Use a MySQL® native interface database connection to import product data from a database table into MATLAB® using a MySQL database. Then, perform a simple data analysis.

Create a MySQL native interface database connection to a MySQL database. The database contains the table `productTable`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
```

```
conn = mysql(datasource,username,password);
```

Import data from the database table `productTable`. The `sqlread` function returns a MATLAB table that contains the product data.

```
tablename = "productTable";
data = sqlread(conn,tablename);
```

Display the first five rows of product data.

```
head(data,5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	1.2597e+05	1003	13	"Victorian Doll"
8	2.1257e+05	1001	5	"Train Set"
7	3.8912e+05	1007	16	"Engine Kit"
2	4.0031e+05	1002	9	"Painting Set"
4	4.0034e+05	1008	21	"Space Cruiser"

Now, import the data using a row filter. The filter condition is that `unitCost` must be less than 15.

```
rf = rowfilter("unitCost");
rf = rf.unitCost < 15;
data = sqlread(conn,tablename,"RowFilter",rf);
```

Again, display the first five rows of product data.

```
head(data,5)
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
9	1.2597e+05	1003	13	"Victorian Doll"
8	2.1257e+05	1001	5	"Train Set"
2	4.0031e+05	1002	9	"Painting Set"
1	4.0034e+05	1001	14	"Building Blocks"
5	4.0046e+05	1005	3	"Tin Soldier"

Close the database connection.

```
close(conn)
```

Import Data from Database Table Using Import Options

Customize import options when importing data from a database table using the MySQL® native interface. Control the import options by creating an `SQLImportOptions` object. Then, customize import options for different database columns. Import data using the `sqlread` function.

This example uses the `patients.xls` file, which contains the columns `Gender`, `Location`, `SelfAssessedHealthStatus`, and `Smoker`. The example also uses a MySQL database version 5.7.22 with the MySQL Connector/C++ driver version 8.0.15.

Create a MySQL native interface database connection to a MySQL database.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";

conn = mysql(datasource,username,password);
```

Load patient information into the MATLAB® workspace.

```
patients = readtable("patients.xls");
```

Create the patients database table using the patient information.

```
tablename = "patients";
sqlwrite(conn,tablename,patients)
```

Create an SQLImportOptions object using the patients database table and the databaseImportOptions function.

```
opts = databaseImportOptions(conn,tablename)
```

```
opts =
  SQLImportOptions with properties:

    ExcludeDuplicates: false
    VariableNamingRule: 'preserve'

    VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    VariableTypes: {'string', 'string', 'double' ... and 7 more}
    SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    FillValues: { <missing>, <missing>, NaN ... and 7 more }
    RowFilter: <unconstrained>

    VariableOptions: Show all 10 VariableOptions
```

Display the current import options for the variables in the SelectedVariableNames property of the SQLImportOptions object.

```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)
```

```
varOpts =
  1x10 SQLVariableImportOptions array with properties:

    Variable Options:

      (1) | (2) | (3) | (4) | (5) | (6) | (7)
      Name: 'LastName' | 'Gender' | 'Age' | 'Location' | 'Height' | 'Weight' | 'Smoker'
      Type: 'string' | 'string' | 'double' | 'string' | 'double' | 'double' | 'logical'
      MissingRule: 'fill' | 'fill' | 'fill' | 'fill' | 'fill' | 'fill' | 'fill'
      FillValue: <missing> | <missing> | NaN | <missing> | NaN | NaN | 0
```

To access sub-properties of each variable, use getoptions

Change the data types for the Gender, Location, Smoker, and SelfAssessedHealthStatus variables using the setoptions function. Because the Gender, Location, and SelfAssessedHealthStatus variables indicate a finite set of repeating values, change their data type to categorical. Because the Smoker variable stores the values 0 and 1, change its data type to double. Then, display the updated import options.

```
opts = setoptions(opts,{'Gender','Location','SelfAssessedHealthStatus'}, ...
  'Type','categorical');
opts = setoptions(opts,'Smoker','Type','double');

varOpts = getoptions(opts,{'Gender','Location','Smoker', ...
  'SelfAssessedHealthStatus'})
```

```
varOpts =
  1x4 SQLVariableImportOptions array with properties:

  Variable Options:
      (1) | (2) | (3) | (4)
      Name: 'Gender' | 'Location' | 'Smoker' | 'SelfAssessedHealthStatus'
      Type: 'categorical' | 'categorical' | 'double' | 'categorical'
      MissingRule: 'fill' | 'fill' | 'fill' | 'fill'
      FillValue: <undefined> | <undefined> | 0 | <undefined>
```

To access sub-properties of each variable, use `getoptions`

Import the `patients` database table using the `sqlread` function, and display the last eight rows of the table.

```
data = sqlread(conn,tablename,opts);
tail(data)
```

LastName	Gender	Age	Location	Height	Weight	Smoker	Sy
"Foster"	Female	30	St. Mary's Medical Center	70	124	0	
"Gonzales"	Male	48	County General Hospital	71	174	0	
"Bryant"	Female	48	County General Hospital	66	134	0	
"Alexander"	Male	25	County General Hospital	69	171	1	
"Russell"	Male	44	VA Hospital	69	188	1	
"Griffin"	Male	49	County General Hospital	70	186	0	
"Diaz"	Male	45	County General Hospital	68	172	1	
"Hayes"	Male	48	County General Hospital	66	177	0	

Display a summary of the imported data. The `sqlread` function applies the import options to the variables in the imported data.

```
summary(data)
```

Variables:

LastName: 200×1 string

Gender: 200×1 categorical

Values:

Female	106
Male	94

Age: 200×1 double

Values:

Min	25
Median	39
Max	50

Location: 200×1 categorical

Values:

County General Hospital	78
St. Mary s Medical Center	48
VA Hospital	74

Height: 200×1 double

Values:

Min	60
Median	67
Max	72

Weight: 200×1 double

Values:

Min	111
Median	142.5
Max	202

Smoker: 200×1 double

Values:

Min	0
Median	0
Max	1

Systolic: 200×1 double

Values:

Min	109
Median	122
Max	138

Diastolic: 200×1 double

Values:

Min	68
Median	81.5
Max	99

SelfAssessedHealthStatus: 200×1 categorical

Values:

Excellent	68
Fair	30
Good	80
Poor	22

Now set the filter condition to import only data for patients older than 40 year and not taller than 68 inches.

```
opts.RowFilter = opts.RowFilter.Age > 40 & opts.RowFilter.Height <= 68
```

```

opts =
  SQLImportOptions with properties:

    ExcludeDuplicates: false
    VariableNamingRule: 'preserve'

    VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    VariableTypes: {'string', 'categorical', 'double' ... and 7 more}
    SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
    FillValues: { <missing>, <undefined>, NaN ... and 7 more }
    RowFilter: Height <= 68 & Age > 40

    VariableOptions: Show all 10 VariableOptions

```

Again, import the `patients` database table using the `sqlread` function, and display a summary of the imported data.

```

data = sqlread(conn,tablename,opts);
summary(data)

```

Variables:

LastName: 48×1 string

Gender: 48×1 categorical

Values:

Female	34
Male	14

Age: 48×1 double

Values:

Min	41
Median	45.5
Max	50

Location: 48×1 categorical

Values:

County General Hospital	26
St. Mary s Medical Center	10
VA Hospital	12

Height: 48×1 double

Values:

Min	62
Median	66
Max	68

Weight: 48×1 double

Values:

Min	119
Median	137
Max	194

Smoker: 48×1 double

Values:

Min	0
Median	0
Max	1

Systolic: 48×1 double

Values:

Min	114
Median	121.5
Max	138

Diastolic: 48×1 double

Values:

Min	68
Median	81.5
Max	96

SelfAssessedHealthStatus: 48×1 categorical

Values:

Excellent	14
Fair	6
Good	20
Poor	8

Delete the `patients` database table using the `execute` function.

```
sqlquery = strcat("DROP TABLE ",tablename);  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Limit Number of Rows in Imported Data

Use a MySQL® native interface database connection to import a limited number of rows of product data from a database table into MATLAB®. Then, sort and filter the rows in the imported data, and perform a simple data analysis.

Create a MySQL® native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the table `productTable`.


```

datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);

```

Import data from the table productTable. Limit the number of rows by setting the 'MaxRows' name-value pair argument to 10. The data table contains the product data.

```

tablename = "productTable";
data = sqlread(conn,tablename,'MaxRows',10);

```

Display the first few rows of product data.

```
data(1:3,:)
```

```

ans=3x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
  _____  _____  _____  _____  _____
           9      1.2597e+05      1003          13      "Victorian Doll"
           8      2.1257e+05      1001           5      "Train Set"
           7      3.8912e+05      1007          16      "Engine Kit"

```

Display the first few product descriptions.

```
data.productDescription(1:3)
```

```

ans = 3x1 string
    "Victorian Doll"
    "Train Set"
    "Engine Kit"

```

Sort the rows in data by the product description column in alphabetical order.

```

column = "productDescription";
data = sortrows(data,column);

```

Display the first few product descriptions after sorting.

```
data.productDescription(1:3)
```

```

ans = 3x1 string
    "Building Blocks"
    "Engine Kit"
    "Painting Set"

```

Close the database connection.

```
close(conn)
```

Retrieve Metadata Information About Imported Data

Retrieve metadata information when importing data from a database table using the MySQL® native interface. Import data using the `sqlread` function and explore the metadata information by using dot notation.

This example uses the `outages.csv` file, which contains outage data. The example also uses a MySQL database version 5.7.22 with the MySQL Connector/C++ driver version 8.0.15.

Create a MySQL® native interface database connection to a MySQL database using the data source name, user name, and password.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Load outage information into the MATLAB® workspace.

```
outages = readtable("outages.csv");
```

Create the `outages` database table using the outage information. Use the 'ColumnType' name-value pair argument to specify the data types of the variables in the MATLAB® table.

```
tablename = "outages";
sqlwrite(conn,tablename,outages, ...
    'ColumnType',["varchar(120)","datetime","numeric(38,16)", ...
    "numeric(38,16)","datetime","varchar(150)"])
```

Import the data into the MATLAB workspace and return metadata information about the imported data.

```
[data,metadata] = sqlread(conn,tablename);
```

View the names of the variables in the imported data.

```
metadata.Properties.RowNames
```

```
ans = 6x1 cell
    {'Region'      }
    {'OutageTime'  }
    {'Loss'        }
    {'Customers'   }
    {'RestorationTime'}
    {'Cause'       }
```

View the data type of each variable in the imported data.

```
metadata.VariableType
```

```
ans = 6x1 cell
    {'string' }
    {'datetime'}
    {'double' }
    {'double' }
    {'datetime'}
    {'string' }
```

View the missing data value for each variable in the imported data.

```
metadata.FillValue
```

```
ans=6x1 cell array
    {1x1 missing}
    {[NaT      ]}
    {[      NaN]}
    {[      NaN]}
    {[NaT      ]}
    {1x1 missing}
```

View the indices of the missing data for each variable in the imported data.

```
metadata.MissingRows
```

```
ans=6x1 cell array
    { 0x1 double}
    { 0x1 double}
    {604x1 double}
    {328x1 double}
    { 29x1 double}
    { 0x1 double}
```

Display the first eight rows of the imported data that contain missing restoration time values. `data` contains restoration time values in the fifth variable. Use the numeric indices to find the rows with missing data.

```
index = metadata.MissingRows{5,1};
nullrestoration = data(index,:);
head(nullrestoration)
```

```
ans=8x6 table
    Region      OutageTime      Loss      Customers      RestorationTime      Cause
    _____ _____
```

Region	OutageTime	Loss	Customers	RestorationTime	Cause
"SouthEast"	23-Jan-2003 00:49:00	530.14	2.1204e+05	NaT	"winter sto
"NorthEast"	18-Sep-2004 05:54:00	0	0	NaT	"equipment
"MidWest"	20-Apr-2002 16:46:00	23141	NaN	NaT	"unknown"
"NorthEast"	16-Sep-2004 19:42:00	4718	NaN	NaT	"unknown"
"SouthEast"	14-Sep-2005 15:45:00	1839.2	3.4144e+05	NaT	"severe sto
"SouthEast"	17-Aug-2004 17:34:00	624.1	1.7879e+05	NaT	"severe sto
"SouthEast"	28-Jan-2006 23:13:00	498.78	NaN	NaT	"energy eme
"West"	20-Jun-2003 18:22:00	0	0	NaT	"energy eme

Delete the outages database table using the `execute` function.

```
sqlstr = "DROP TABLE ";
sqlquery = strcat(sqlstr,tablename);
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

opts — Database import options

SQLImportOptions object

Database import options, specified as an SQLImportOptions object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `data =`

`sqlread(conn, 'inventoryTable', 'Catalog', 'toy_store', 'MaxRows', 5)` imports five rows of data from the database table `inventoryTable` stored in the `toy_store` catalog.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: string | char

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of 'MaxRows' and a positive numeric scalar. By default, the `sqlread` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: `'MaxRows', 10`

Data Types: double

VariableNamingRule — Variable naming rule

"preserve" (default) | "modify"

Variable naming rule, specified as the comma-separated pair consisting of 'VariableNamingRule' and one of these values:

- "preserve" — Preserve most variable names when the `sqlread` function imports data. For details, see the Limitations section.
- "modify" — Remove non-ASCII characters from variable names when the `sqlread` function imports data.

Example: 'VariableNamingRule', "modify"

Data Types: string

RowFilter — Row filter condition

<unconstrained> (default) | `matlab.io.RowFilter` object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; sqlread(conn,tablename,"RowFilter",rf)`

Output Arguments

data — Imported data

table

Imported data, returned as a table. The rows of the table correspond to the rows in the database table `tablename`. The variables in the table correspond to each column in the database table.

If the database table contains no data to import, then `data` is an empty table.

When you import data, the `sqlread` function converts the data type of each column from the MySQL database to the MATLAB data type. This table maps the data type of a database column to the converted MATLAB data type.

MySQL Data Type	MATLAB Data Type
BIT	logical
TINYINT	double
SMALLINT	double
BIGINT	double
REAL	double
DOUBLE	double
DECIMAL	double
NUMERIC	double
CHAR	string
VARCHAR	string
LONGVARCHAR	string
TIMESTAMP	datetime
DATE	datetime

MySQL Data Type	MATLAB Data Type
TIME	duration
YEAR	double
ENUM	categorical
JSON	char

metadata — Metadata information

table

Metadata information, returned as a table with these variables.

Variable Name	Variable Description	Variable Data Type
VariableType	Data type of each variable in the imported data	Cell array of character vectors
FillValue	Value of missing data for each variable in the imported data	Cell array of missing data values
MissingRows	Indices for each occurrence of missing data in each variable of the imported data	Cell array of numeric indices

By default, the `sqlread` function imports text data as a character vector and numeric data as a double. `FillValue` is an empty character array (for text data) or NaN (for numeric data) by default. To change the missing data value to another value, use the `SQLImportOptions` object.

The `RowNames` property of the `metadata` table contains the names of the variables in the imported data.

Limitations

- The `sqlread` function returns an error when you use the `VariableNamingRule` name-value argument with the `SQLImportOptions` object `opts`.
- When the `VariableNamingRule` name-value pair argument is set to the value "modify":
 - The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the table data type.
 - The length of each variable name must be less than the number returned by `namelengthmax`.
- The `sqlread` function returns an error if you specify the `RowFilter` name-value argument with the `SQLImportOptions` object `opts`. It is ambiguous which of the `RowFilter` object to use in this case, especially if the filter conditions are different.

Version History

Introduced in R2020b

R2023a: Selectively import rows of data based on filter condition

You can use the `RowFilter` name-value argument to selectively import rows of data from a database table.

See Also

Functions

sqlfind | fetch | sqlinnerjoin | sqlouterjoin | mysql | close | execute | databaseImportOptions | setoptions | getoptions | reset

Topics

“Import Data from MySQL Database Table” on page 6-6

“Customize Options for Importing Data from Database into MATLAB Using MySQL Native Interface” on page 6-8

sqlupdate

Package: database.mysql

Update rows in MySQL database table

Syntax

```
sqlupdate(conn,tablename,data,filter)
sqlupdate( ____,Name,Value)
```

Description

`sqlupdate(conn,tablename,data,filter)` updates rows in the MySQL database table (`tablename`) with the rows from the MATLAB table (`data`) based on filter conditions (`filter`).

`sqlupdate(____,Name,Value)` specifies additional options using one or more name-value arguments with any of the previous input argument combinations. For example, `Catalog = "cat"` updates data from a database table stored in the "cat" catalog.

Examples

Update Database Rows

Update rows in the MySQL native interface database based on filter conditions specified with row filters.

This example uses the `patients.xls` file, which contains the columns `LastName`, `Gender`, `Age`, `Location`, `Height`, `Weight`, `Smoker`, `Systolic`, `Diastolic`, and `SelfAssessedHealthStatus`. The example also uses a MySQL database version 5.7.22 with the MySQL Connector/C++ driver version 8.0.15.

Create a MySQL native interface database connection to a MySQL database.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
```

```
conn = mysql(datasource,username,password);
```

Load patient information into the MATLAB workspace.

```
patients = readtable("patients.xls");
```

Create the `patients` database table using the patient information.

```
tablename = "patients";
sqlwrite(conn,tablename,patients)
```

Use the SQL ALTER statement to add the column `HighRisk` to the table `patients`.


```
sqlquery = 'ALTER TABLE patients ADD HighRisk bit';
execute(conn,sqlquery)
```

Import the patients database table using the `sqlread` function, and return metadata information about the imported data.

```
[data,metadata] = sqlread(conn,tablename);
```

Display the first 10 rows of the table. In MATLAB, all the values in the `HighRisk` column appear as `false`.

```
head(data,10)
```

LastName	Gender	Age	Location	Height	Weight	Smoker
"Smith"	"Male"	38	"County General Hospital"	71	176	true
"Johnson"	"Male"	43	"VA Hospital"	69	163	false
"Williams"	"Female"	38	"St. Mary's Medical Center"	64	131	false
"Jones"	"Female"	40	"VA Hospital"	67	133	false
"Brown"	"Female"	49	"County General Hospital"	64	119	false
"Davis"	"Female"	46	"St. Mary's Medical Center"	68	142	false
"Miller"	"Female"	33	"VA Hospital"	64	142	true
"Wilson"	"Male"	40	"VA Hospital"	68	180	false
"Moore"	"Male"	28	"St. Mary's Medical Center"	68	183	false
"Taylor"	"Female"	31	"County General Hospital"	66	132	false

Displaying the metadata shows that the values are `NULL` (missing elements) in the database.

```
metadata
```

```
metadata=11x3 table
```

	VariableType	FillValue	MissingRows
LastName	{'string' }	{[<missing>]}	{ 0x1 double}
Gender	{'string' }	{[<missing>]}	{ 0x1 double}
Age	{'double' }	{[NaN]}	{ 0x1 double}
Location	{'string' }	{[<missing>]}	{ 0x1 double}
Height	{'double' }	{[NaN]}	{ 0x1 double}
Weight	{'double' }	{[NaN]}	{ 0x1 double}
Smoker	{'logical' }	{[0]}	{ 0x1 double}
Systolic	{'double' }	{[NaN]}	{ 0x1 double}
Diastolic	{'double' }	{[NaN]}	{ 0x1 double}
SelfAssessedHealthStatus	{'string' }	{[<missing>]}	{ 0x1 double}
HighRisk	{'logical' }	{[0]}	{100x1 double}

Now, identify patients who are considered high risk for developing some hypothetical health issue based on their age and their smoker status. First, create a table containing the new data to write to the database. This table requires only 1 (true) and 0 (false) values.

```
t = table([1;0], "VariableNames", "HighRisk");
head(t)
```

```
HighRisk
```

```
1
0
```

Create a row filter using the filter condition that a patient must be older than 35 years and a smoker to be considered high-risk.

```
rf = rowfilter(["Age", "Smoker"]);
rf = rf.Age > 35 & rf.Smoker == 1
```

```
rf =
  RowFilter with constraints:

    Age > 35 & Smoker == 1

  VariableNames: Age, Smoker
```

Update the `HighRisk` column using this filter to set the values to 1 (true) and using the `~rf` value of the filter to set the value to 0 (false).

```
sqlupdate(conn, "patients", t, {rf;~rf});
```

Again, import the `patients` database table using the `sqlread` function, and display the first 10 rows.

```
data = sqlread(conn, tablename);
head(data, 10)
```

LastName	Gender	Age	Location	Height	Weight	Smoker
"Smith"	"Male"	38	"County General Hospital"	71	176	true
"Johnson"	"Male"	43	"VA Hospital"	69	163	false
"Williams"	"Female"	38	"St. Mary's Medical Center"	64	131	false
"Jones"	"Female"	40	"VA Hospital"	67	133	false
"Brown"	"Female"	49	"County General Hospital"	64	119	false
"Davis"	"Female"	46	"St. Mary's Medical Center"	68	142	false
"Miller"	"Female"	33	"VA Hospital"	64	142	true
"Wilson"	"Male"	40	"VA Hospital"	68	180	false
"Moore"	"Male"	28	"St. Mary's Medical Center"	68	183	false
"Taylor"	"Female"	31	"County General Hospital"	66	132	false

Delete the `patients` database table using the `execute` function.

```
sqlquery = strcat("DROP TABLE ", tablename);
execute(conn, sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

data — Updated data

MATLAB table

Updated data, specified as a MATLAB table. The table can contain one or more rows with updated data. The names of the variables in the table must be a subset of the column names of the database table.

Example: `data = table([1;0], "VariableNames", "NewName")`

Data Types: table

filter — Row filter condition

matlab.io.RowFilter object | cell array of matlab.io.RowFilter objects

Row filter condition, specified as a `matlab.io.RowFilter` object or cell array of `matlab.io.RowFilter` objects. Filters determine which database rows `sqlupdate` must update with which data. If multiple database rows match a filter, `sqlupdate` updates them with the same data. If a single database row matches multiple filters, its final state matches the data corresponding to the last matching filter.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5;`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `sqlupdate(conn, 'inventoryTable', data, rf, Catalog = "toy_store", Schema = "dbo")` updates the database `inventoryTable` stored in the `toy_store` catalog and the `dbo` schema.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: string | char

Schema — Database schema name

string scalar | character vector

Database schema name, specified as a string scalar or character vector. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Example: `Schema = "dbo"`

Data Types: `string` | `char`

Version History

Introduced in R2023a

See Also

`sqlfind` | `fetch` | `sqlread` | `sqlinnerjoin` | `sqlouterjoin` | `mysql` | `close` | `execute` | `databaseImportOptions` | `setoptions` | `getoptions` | `reset`

sqlwrite

Package: database.mysql

Insert MATLAB data into MySQL database table

Syntax

```
sqlwrite(conn,tablename,data)
sqlwrite(conn,tablename,data,Name,Value)
```

Description

`sqlwrite(conn,tablename,data)` inserts data from a MATLAB table into a database table. If the table exists in the database, this function appends the data from the MATLAB table as rows in the existing database table. If the table does not exist in the database, this function creates a table with the specified table name and then inserts the data as rows in the new table. This syntax is the equivalent of executing SQL statements that contain the CREATE TABLE and INSERT INTO ANSI SQL syntaxes.

`sqlwrite(conn,tablename,data,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, 'Catalog','toy_store' inserts data into a database table that is located in the database catalog named toy_store.

Examples

Append Data into Existing Table Using MySQL Native Interface

Use a MySQL® native interface database connection to append product data from a MATLAB® table into an existing table in a MySQL database.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the table `productTable`.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

To view the existing database table `productTable` before appending data, import its contents into MATLAB and display the last few rows.

```
tablename = "productTable";
rows = sqlread(conn,tablename);
tail(rows,3)
```

```
ans=3x5 table
    productNumber    stockNumber    supplierNumber    unitCost    productDescription
    _____    _____    _____    _____    _____
             6      4.0088e+05      1004              8      "Sail Boat"
```

3	4.01e+05	1009	17	"Slinky"
10	8.8865e+05	1006	24	"Teddy Bear"

Create a MATLAB table that contains the data for one product.

```
data = table(30,500000,1000,25,"Rubik's Cube", ...
    'VariableNames',["productNumber" "stockNumber" ...
    "supplierNumber" "unitCost" "productDescription"]);
```

Append the product data into the database table productTable.

```
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB again and display the last few rows. The results contain a new row for the inserted product.

```
rows = sqlread(conn,tablename);
tail(rows,4)
```

```
ans=4x5 table
    productNumber    stockNumber    supplierNumber    unitCost    productDescription
    _____    _____    _____    _____    _____
         6         4.0088e+05         1004             8         "Sail Boat"
         3         4.01e+05         1009             17         "Slinky"
        10         8.8865e+05         1006             24         "Teddy Bear"
        30             5e+05         1000             25         "Rubik's Cube"
```

Close the database connection.

```
close(conn)
```

Insert Data into New Table

Use a MySQL® native interface database connection to insert product data from MATLAB® into a new table in a MySQL database.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password. The database contains the table productTable.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Create a MATLAB table that contains data for two products.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"],'VariableNames',["productNumber" ...
    "stockNumber" "supplierNumber" "unitCost" "productDescription"]);
```

Insert the product data into a new database table named toyTable.

```
tablename = "toyTable";
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
  _____  _____  _____  _____  _____
           30           5e+05           1000           25           "Rubik's Cube"
           40           6e+05           2000           30           "Doll House"
```

Close the database connection.

```
close(conn)
```

Specify Column Types When Inserting Data into New Table

Use a MySQL® native interface database connection to insert product data from MATLAB® into a new table in a MySQL database. Specify the data types of the columns in the new database table.

Create a MySQL native interface database connection to a MySQL database using the data source name, user name, and password.

```
datasource = "MySQLNative";
username = "root";
password = "matlab";
conn = mysql(datasource,username,password);
```

Create a MATLAB table that contains data for two products.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
  ["Rubik's Cube";"Doll House"],'VariableNames',["productNumber" ...
  "stockNumber" "supplierNumber" "unitCost" "productDescription"]);
```

Insert the product data into a new database table named toyTable. Use the 'ColumnType' name-value pair argument and a string array to specify the data types of all the columns in the database table.

```
tablename = "toyTable";
coltypes = ["numeric" "numeric" "numeric" "numeric" "varchar(255)"];
sqlwrite(conn,tablename,data,'ColumnType',coltypes)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
  productNumber  stockNumber  supplierNumber  unitCost  productDescription
  _____  _____  _____  _____  _____
```

30	5e+05	1000	25	"Rubik's Cube"
40	6e+05	2000	30	"Doll House"

Close the database connection.

```
close(conn)
```

Input Arguments

conn — MySQL native interface database connection

connection object

MySQL native interface database connection, specified as a connection object.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

data — Data to insert

table

Data to insert into a database table, specified as a table.

The valid data types in a MATLAB table are:

- Numeric array
- Cell array of numeric arrays
- Cell array of character vectors
- String array
- Datetime array
- Duration array
- Logical array
- Cell array of logical arrays

The numeric array can contain these data types:

- int8
- uint8
- int16
- uint16
- int32
- uint32
- int64

- uint64
- single
- double

For date and time data, supported formats are:

- Date — 'yyyy-MM-dd'
- Time — 'hh:mm:ss'
- Timestamp — 'yyyy-MM-dd HH:mm:ss'

If the date and time data is specified in an invalid format, then the `sqlwrite` function automatically converts the data to a supported format.

If the cell array of character vectors or string array is specified in an invalid format, then the `sqlwrite` function enables the database driver to check the format. If the format is unexpected, then the database driver throws an error.

You can insert data in an existing database table or a new database table. The data types of variables in `data` vary depending on whether the database table exists. For valid data types, see “Data Types for Existing Table” on page 11-0 and “Data Types for New Table” on page 11-0 .

Note The `sqlwrite` function supports only the `table` data type for the data input argument. To insert data stored in a structure, cell array, or numeric matrix, convert the data to a `table` by using the `struct2table`, `cell2table`, and `array2table` functions, respectively.

To insert missing data, see “Accepted Missing Data” on page 11-0 .

Example: `table([10;20],{'M';'F'})`

Data Types for Existing Table

The variable names of the MATLAB table must match the column names in the database table. The `sqlwrite` function is case-sensitive.

When you insert data into a database table, use the data types shown in the following table to ensure that the data has the correct data type. This table matches the valid data types of the MATLAB table variable to the data types of the database column. For example, when you insert data into a database column that has the `BIT` data type, ensure that the corresponding variable in the MATLAB table is a logical array or cell array of logical arrays.

Data Type of MATLAB Table Variable	Data Type of Existing Database Column
Numeric array or cell array of numeric arrays	<ul style="list-style-type: none"> • INTEGER • SMALLINT • DECIMAL • NUMERIC • FLOAT • REAL • DOUBLE PRECISION

Data Type of MATLAB Table Variable	Data Type of Existing Database Column
Cell array of character vectors, string array, datetime array, or duration array	<ul style="list-style-type: none"> • DATE • TIME • TIMESTAMP
Logical array or cell array of logical arrays	BIT
Cell array of character vectors or string array	<ul style="list-style-type: none"> • CHAR • VARCHAR

Data Types for New Table

The specified table name for the new database table must be unique across all tables in the database.

The valid data types in a MATLAB table are:

- Numeric array
- Cell array of character vectors
- String array
- Datetime array
- Duration array
- Logical array

The `sqlwrite` function ignores any invalid data types and inserts only the valid variables from MATLAB as columns in a new database table.

The `sqlwrite` function converts the data type of the variable into the default data type of the column in the database table. The following table matches the valid data types of the MATLAB table variable to the default data types of the database column.

Data Type of MATLAB Table Variable	Default Data Type of Database Column
int8 array	TINYINT
int16 array	SMALLINT
int32 array	INTEGER
int64 array	BIGINT
logical array	BIT
single or double array	NUMERIC
datetime array	TIMESTAMP
duration array	TIME
cell array of character vectors or string array	VARCHAR Note The size of this column equals the sum of the maximum length of a string in the string array and 100.

To specify database-specific column data types instead of the defaults, use the 'ColumnType' name-value pair argument. For example, you can specify 'ColumnType', 'bigint' to create a BIGINT column in the new database table.

Also, using the 'ColumnType' name-value pair argument, you can specify other data types that are not in the default list. For example, to insert images, specify 'ColumnType', "image".

Accepted Missing Data

The accepted missing data for inserting data into a database depends on the data type of the MATLAB table variable and the data type of the column in the database. The following table matches the data type of the MATLAB table variable to the data type of the database column and specifies the accepted missing data to use in each case.

Data Type of MATLAB Table Variable	Data Type of Database Column	Accepted Missing Data
datetime array	Date or Timestamp	NaN
duration array	Time	NaN
double or single array or cell array of double or single arrays	Numeric	NaN, [], or ''
cell array of character vectors	Date or Timestamp	'NaN' or ''
cell array of character vectors	Time	'NaN' or ''
cell array of character vectors	Char, Varchar, or other text data type	''
string array	Date or Timestamp	"", "NaN", or missing
string array	Time	"", "NaN", or missing
string array	Char, Varchar, or other text data type	missing

Data Types: table

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `sqlwrite(conn,"tablename",data,'ColumnType',["numeric" "timestamp" "image"])` inserts data into a new database table named `tablename` by specifying data types for all columns in the new database table.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: string | char

ColumnType — Database column types

character vector | string scalar | cell array of character vectors | string array

Database column types, specified as the comma-separated pair consisting of 'ColumnType' and a character vector, string scalar, cell array of character vectors, or string array. Use this name-value pair argument to define custom data types for the columns in a database table. Specify a column type for each column in the table.

Example: 'ColumnType', ["numeric" "varchar(400)"]

Data Types: char | string | cell

Version History

Introduced in R2020b

See Also

sqlread | mysql | close | cell2table | array2table | struct2table

Topics

“Import Data from MySQL Database Table” on page 6-6

“Delete Data from Database Using MySQL Native Interface” on page 6-19

“Roll Back Data in Database Using MySQL Native Interface” on page 6-16

connection

PostgreSQL native interface database connection

Description

Create a connection to a PostgreSQL database using the PostgreSQL native interface. Configure a PostgreSQL native interface data source using the `databaseConnectionOptions` function.

Creation

Create a `connection` object by using the `postgresql` function.

Properties

DataSource — Data source name

string scalar

This property is read-only.

Data source name, specified as a string scalar.

Example: "PostgreSQLDataSource"

Data Types: string

Database — Database name

"" (default) | string scalar

This property is read-only.

Database name, specified as a string scalar.

If you use the 'DatabaseName' name-value pair argument of the `postgresql` function, the `postgresql` function sets the `Database` property of the `connection` object to the specified value.

Example: "toystore_doc"

Data Types: string

Server — Server name

localhost (default) | string scalar

This property is read-only.

Server name, specified as a string scalar.

If you use the 'Server' name-value pair argument of the `postgresql` function, the `postgresql` function sets the `Server` property of the `connection` object to the specified value.

Example: "dbtb00"

Data Types: `string`

PortNumber — Port number

5432 (default) | numeric scalar

This property is read-only.

Port number, specified as a numeric scalar.

If you use the 'PortNumber' name-value pair argument of the `postgresql` function, the `postgresql` function sets the `PortNumber` property of the connection object to the specified value.

Example: 5432

Data Types: `double`

UserName — User name

"" (default) | string scalar

This property is read-only.

User name, specified as a string scalar.

Data Types: `string`

DefaultCatalog — Default catalog

"" (default) | string scalar

This property is read-only.

Default catalog, specified as a string scalar.

Example: `"toy_store"`

Data Types: `string`

Catalogs — Catalogs in database

"" (default) | string array

This property is read-only.

Catalogs in the database, specified as a string array.

Example: `["information", "postgresql"]`

Data Types: `string`

Schemas — Schemas in database

"" (default) | string array

This property is read-only.

Schemas in the database, specified as a string array.

Example: `["information_schema", "toys"]`

Data Types: `string`

AutoCommit — Flag to autocommit transactions

string scalar

Flag to autocommit transactions, specified as one of these values:

- "on" — Database transactions are automatically committed to the database.
- "off" — Database transactions must be committed to the database manually.

You can set this property by using dot notation.

LoginTimeout — Login timeout

0 (default) | positive numeric scalar

This property is read-only.

Login timeout, specified as a positive numeric scalar. The login timeout specifies the number of seconds that the driver waits while trying to connect to a database before throwing an error.

When no login timeout for the connection attempt is specified, the value is 0.

When a login timeout is not supported by the database, the value is -1.

Data Types: double

MaxDatabaseConnections — Maximum number of database connections

-1 (default) | positive numeric scalar

This property is read-only.

Maximum number of database connections, specified as a positive numeric scalar.

When the database has no upper limit to the maximum number of database connections, the value is 0.

When a maximum number of database connections is not supported by the database, the value is -1.

Data Types: double

DatabaseProductName — Database product name

"" (default) | string scalar

This property is read-only.

Database product name, specified as a string scalar.

When the database connection is invalid, the value is an empty string scalar "".

Example: "PostgreSQL"

Data Types: string

DatabaseProductVersion — Database product version

"" (default) | string scalar

This property is read-only.

Database product version, specified as a string scalar.

When the database connection is invalid, the value is an empty string scalar "".

Example: "9.4.5"

Data Types: string

DriverName — Driver name

"" (default) | string scalar

This property is read-only.

Driver name of the PostgreSQL driver, specified as a string scalar.

When the database connection is invalid, the value is an empty string scalar "".

Example: "libpq"

Data Types: string

DriverVersion — Driver version

"" (default) | string scalar

This property is read-only.

Driver version of the PostgreSQL driver, specified as a string scalar.

When the database connection is invalid, the value is an empty string scalar "".

Example: "10.12"

Data Types: string

Object Functions

Manage PostgreSQL Database Connection

close Close PostgreSQL native interface database connection

isopen Determine if PostgreSQL native interface database connection is open

Import Data from PostgreSQL Database

fetch Import results of SQL statement in PostgreSQL database into MATLAB

sqlinnerjoin Inner join between two PostgreSQL database tables

sqlouterjoin Outer join between two PostgreSQL database tables

sqlfind Find information about all table types in PostgreSQL database

sqlread Import data into MATLAB from PostgreSQL database table

executeSQLScript Execute SQL script on PostgreSQL database

Export Data to PostgreSQL Database

sqlwrite Insert MATLAB data into PostgreSQL database table

PostgreSQL Database Operations

execute Execute SQL statement using PostgreSQL native interface database connection

commit Make changes to PostgreSQL database permanent

rollback Undo changes to PostgreSQL database

`sqlupdate` Update rows in PostgreSQL database table

Examples

Connect to PostgreSQL Database Using PostgreSQL Native Interface

Create a PostgreSQL native interface connection to a PostgreSQL database. Then, import data from the database into MATLAB® and perform simple data analysis. Close the database connection.

This example assumes that you are connecting to a PostgreSQL database version 9.405 using the libpq driver version 10.12.

Connect to the database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";

conn = postgresql(datasource,username,password)

conn =
  connection with properties:

      DataSource: "PostgreSQLDataSource"
      UserName: "dbdev"

  Database Properties:

      AutoCommit: "on"
      LoginTimeout: 0
      MaxDatabaseConnections: 100

  Catalog and Schema Information:

      DefaultCatalog: "toystore_doc"
      Catalogs: "toystore_doc"
      Schemas: ["pg_toast", "pg_temp_1", "pg_toast_temp_1" ... and 3 more]

  Database and Driver Information:

      DatabaseProductName: "PostgreSQL"
      DatabaseProductVersion: "9.405"
      DriverName: "libpq"
      DriverVersion: "10.12"
```

The property sections of the `connection` object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `sqlread` function. Display the first three rows of data.

```
tablename = "inventoryTable";  
data = sqlread(conn,tablename);  
head(data,3)
```

```
ans=3x4 table  
  productnumber  quantity  price  inventorydate  
-----  
           1      1700    14.5  "2014-09-23 09:38:34"  
           2      1200     9    "2014-07-08 22:50:45"  
           3       356    17    "2014-05-14 07:14:28"
```

Determine the highest product quantity from the table.

```
max(data.quantity)
```

```
ans = 9000
```

Close the database connection conn.

```
close(conn)
```

Version History

Introduced in R2020b

See Also

[close](#) | [fetch](#) | [sqlread](#) | [sqlupdate](#)

Topics

“Import Data from PostgreSQL Database Table” on page 7-11

postgresql

Create PostgreSQL native interface database connection

Syntax

```
conn = postgresql(datasource,username,password)
conn = postgresql(username,password,Name,Value)
```

Description

`conn = postgresql(datasource,username,password)` creates a PostgreSQL native interface database connection using the specified data source, user name, and password. `conn` is a connection object.

`conn = postgresql(username,password,Name,Value)` creates a PostgreSQL native interface database connection using the specified user name and password, with additional options specified by one or more name-value pair arguments. For example, "Server", "dbtb00" specifies the database server name as dbtb00.

Examples

Connect to PostgreSQL Database Using PostgreSQL Native Interface

Create a PostgreSQL native interface connection to a PostgreSQL database. Then, import data from the database into MATLAB® and perform simple data analysis. Close the database connection.

This example assumes that you are connecting to a PostgreSQL database version 9.405 using the libpq driver version 10.12.

Connect to the database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";

conn = postgresql(datasource,username,password)

conn =
  connection with properties:

      DataSource: "PostgreSQLDataSource"
      UserName: "dbdev"

  Database Properties:

      AutoCommit: "on"
      LoginTimeout: 0
      MaxDatabaseConnections: 100

  Catalog and Schema Information:
```

```

DefaultCatalog: "toystore_doc"
Catalogs: "toystore_doc"
Schemas: ["pg_toast", "pg_temp_1", "pg_toast_temp_1" ... and 3 more]

```

Database and Driver Information:

```

DatabaseProductName: "PostgreSQL"
DatabaseProductVersion: "9.405"
DriverName: "libpq"
DriverVersion: "10.12"

```

The property sections of the `connection` object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `sqlread` function. Display the first three rows of data.

```

tablename = "inventoryTable";
data = sqlread(conn,tablename);
head(data,3)

```

```

ans=3x4 table
   productnumber   quantity   price   inventorydate
   _____   _____   _____   _____
           1           1700        14.5   "2014-09-23 09:38:34"
           2           1200         9     "2014-07-08 22:50:45"
           3            356         17     "2014-05-14 07:14:28"

```

Determine the highest product quantity from the table.

```
max(data.quantity)
```

```
ans = 9000
```

Close the database connection `conn`.

```
close(conn)
```

Connect to PostgreSQL Database Using PostgreSQL Native Interface and Additional Options

Create a PostgreSQL native interface connection to a PostgreSQL database using name-value pair arguments. Then, import data from the database into MATLAB® and perform simple data analysis. Close the database connection.

This example assumes that you are connecting to a PostgreSQL database version 9.405 using the `libpq` driver version 10.12.

Connect to the database using the user name and password shown. Specify the database server name `dbtb00`, database name `toystore_doc`, and port number 5432 by setting the corresponding name-value pair arguments.

```
username = "dbdev";
password = "matlab";

conn = postgresql(username,password,'Server','dbtb00', ...
    'DatabaseName','toystore_doc','PortNumber',5432)

conn =
    connection with properties:

        Database: "toystore_doc"
        UserName: "dbdev"

    Database Properties:

        AutoCommit: "on"
        LoginTimeout: 0
        MaxDatabaseConnections: 100

    Catalog and Schema Information:

        DefaultCatalog: "toystore_doc"
        Catalogs: "toystore_doc"
        Schemas: ["pg_toast", "pg_temp_1", "pg_toast_temp_1" ... and 3 more]

    Database and Driver Information:

        DatabaseProductName: "PostgreSQL"
        DatabaseProductVersion: "9.405"
        DriverName: "libpq"
        DriverVersion: "10.12"
```

The property sections of the `connection` object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `sqlread` function. Display the first three rows of data.

```
tablename = "inventoryTable";
data = sqlread(conn,tablename);
head(data,3)
```

ans=3x4 table

productnumber	quantity	price	inventorydate
1	1700	14.5	"2014-09-23 09:38:34"
2	1200	9	"2014-07-08 22:50:45"
3	356	17	"2014-05-14 07:14:28"

Determine the highest product quantity from the table.

```
max(data.quantity)
```

```
ans = 9000
```

Close the database connection `conn`.

```
close(conn)
```

Input Arguments

datasource — Data source name

character vector | string scalar

Data source name, specified as a character vector or string scalar. Specify the name of an existing data source.

Example: "myDataSource"

Data Types: char | string

username — User name

character vector | string scalar

User name required to access the database, specified as a character vector or string scalar. If no user name is required, specify an empty value "".

Data Types: char | string

password — Password

character vector | string scalar

Password required to access the database, specified as a character vector or string scalar. If no password is required, specify an empty value "".

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `conn =`

```
postgresql(username,password,"Server","dbtb00","PortNumber",5432,"DatabaseName","toystore_doc")
```

 creates a PostgreSQL native interface database connection using the database server `dbtb00`, port number 5432, and database name `toystore_doc`.

Server — Database server name

"localhost" (default) | string scalar | character vector

Database server name or address, specified as the comma-separated pair consisting of 'Server' and a string scalar or character vector.

Example: "dbtb00"

Data Types: char | string

PortNumber – Port number

5432 (default) | numeric scalar

Port number, specified as the comma-separated pair consisting of 'PortNumber' and a numeric scalar.

Example: 5432

Data Types: double

DatabaseName – Database name

"" (default) | string scalar | character vector

Database name, specified as the comma-separated pair consisting of 'DatabaseName' and a string scalar or character vector. If you do not specify a database name, the `postgresql` function connects to the default database on the database server.

Example: "toystore_doc"

Data Types: char | string

Version History

Introduced in R2020b

See Also

`databaseConnectionOptions` | `close` | `isopen` | `sqlread`

Topics

“Configure PostgreSQL Native Interface Data Source” on page 7-2

“Import Data from PostgreSQL Database Table” on page 7-11

“Insert Data into Database Table Using PostgreSQL Native Interface” on page 7-19

close

Package: database.postgre

Close PostgreSQL native interface database connection

Syntax

```
close(conn)
```

Description

`close(conn)` closes and invalidates the PostgreSQL native interface database connection to free up database and driver resources.

Examples

Connect to PostgreSQL Database Using PostgreSQL Native Interface

Create a PostgreSQL native interface connection to a PostgreSQL database. Then, import data from the database into MATLAB® and perform simple data analysis. Close the database connection.

This example assumes that you are connecting to a PostgreSQL database version 9.405 using the libpq driver version 10.12.

Connect to the database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";  
username = "dbdev";  
password = "matlab";
```

```
conn = postgresql(datasource,username,password)
```

```
conn =
```

```
connection with properties:
```

```
DataSource: "PostgreSQLDataSource"  
UserName: "dbdev"
```

```
Database Properties:
```

```
AutoCommit: "on"  
LoginTimeout: 0  
MaxDatabaseConnections: 100
```

```
Catalog and Schema Information:
```

```
DefaultCatalog: "toystore_doc"  
Catalogs: "toystore_doc"  
Schemas: ["pg_toast", "pg_temp_1", "pg_toast_temp_1" ... and 3 more]
```

```
Database and Driver Information:
```



```

DatabaseProductName: "PostgreSQL"
DatabaseProductVersion: "9.405"
DriverName: "libpq"
DriverVersion: "10.12"

```

The property sections of the `connection` object are:

- **Database Properties** — Information about the database configuration
- **Catalog and Schema Information** — Names of catalogs and schemas in the database
- **Database and Driver Information** — Names and versions of the database and driver

Import all data from the table `inventoryTable` into MATLAB using the `sqlread` function. Display the first three rows of data.

```

tablename = "inventoryTable";
data = sqlread(conn,tablename);
head(data,3)

```

```

ans=3x4 table
   productnumber   quantity   price   inventorydate
   _____   _____   _____   _____
           1           1700      14.5   "2014-09-23 09:38:34"
           2           1200           9   "2014-07-08 22:50:45"
           3            356          17   "2014-05-14 07:14:28"

```

Determine the highest product quantity from the table.

```
max(data.quantity)
```

```
ans = 9000
```

Close the database connection `conn`.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

Version History

Introduced in R2020b

See Also

postgresql | fetch

Topics

“Configure PostgreSQL Native Interface Data Source” on page 7-2

“Import Data from PostgreSQL Database Table” on page 7-11

commit

Package: database.postgre

Make changes to PostgreSQL database permanent

Syntax

```
commit(conn)
```

Description

`commit(conn)` makes changes to the database connection `conn` permanent, specifically any changes made since the last `commit` or `rollback` function has been run. To use the `commit` function, you must set the `AutoCommit` property of the connection object to `off`.

Examples

Commit Data to PostgreSQL Database

Use a PostgreSQL native interface database connection to insert product data from MATLAB® into a new table in a PostgreSQL database. Then, commit the changes to the database.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Allow manual committing of changes to the database by setting the `AutoCommit` property to `off`.

```
conn.AutoCommit = "off";
```

Create a MATLAB table that contains data for two products.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"],'VariableNames',["productNumber" ...
    "stockNumber" "supplierNumber" "unitCost" "productDescription"]);
```

Insert the product data into a new database table `toytable`.

```
tablename = "toytable";
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
  productnumber  stocknumber  suppliernumber  unitcost  productdescription
-----
           30      5e+05      1000           25      "Rubik's Cube"
           40      6e+05      2000           30      "Doll House"
```

Commit the changes to the database.

```
commit(conn)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

Version History

Introduced in R2020b

See Also

postgresql | execute | rollback | sqlwrite

Topics

“Insert Data into Database Table Using PostgreSQL Native Interface” on page 7-19

“Create Table and Add Column Using PostgreSQL Native Interface” on page 7-23

“Delete Data from Database Using PostgreSQL Native Interface” on page 7-24

“Roll Back Data in Database Using PostgreSQL Native Interface” on page 7-21

SQLConnectionOptions

Define PostgreSQL native interface database connection options

Description

Create database connection options for a PostgreSQL native interface connection.

First, create an `SQLConnectionOptions` object, set the connection options, test the connection, and save the data source. Then, create a PostgreSQL native interface connection using the saved data source. The connection options include the options required to make a database connection. You can also define additional connection options for a specific database driver.

Creation

Create an `SQLConnectionOptions` object using the `databaseConnectionOptions` function.

Properties

DataSourceName — Data source name

string scalar

Data source name, specified as a string scalar. You can use the data source name in the `postgresql` function to create a database connection for the PostgreSQL native interface.

Example: "PostgreSQLDataSource"

Data Types: string

Vendor — Database vendor

string scalar

This property is read-only.

Database vendor, specified as a string scalar. Specify this property using the `vendor` input argument in the `databaseConnectionOptions` function. After the `SQLConnectionOptions` object exists, you cannot set this property to another value.

Example: "PostgreSQL"

Data Types: string

DatabaseName — Database name

string scalar

Database name, specified as a string scalar. Set this property using the `setoptions` function.

Example: "toystore_doc"

Data Types: string

Server — Database server name or address`"localhost"` (default) | string scalar

Database server name or address, specified as a string scalar. Set this property using the `setoptions` function.

Data Types: `string`

PortNumber — Server port number where the server is listening`5432` (default) | numeric scalar

Server port number where the server is listening, specified as a numeric scalar. Set this property using the `setoptions` function.

Data Types: `double`

Object Functions

<code>rmoptions</code>	Remove PostgreSQL native interface connection options
<code>saveAsDataSource</code>	Save PostgreSQL native interface data source
<code>setoptions</code>	Set PostgreSQL native interface connection options
<code>reset</code>	Reset PostgreSQL native interface connection options to defaults
<code>testConnection</code>	Test PostgreSQL native interface database connection

Examples**Create PostgreSQL Native Interface Data Source**

Create, configure, test, and save a PostgreSQL native interface data source for a PostgreSQL database.

Create a PostgreSQL native interface data source for a PostgreSQL native interface database connection.

```
vendor = "PostgreSQL";  
opts = databaseConnectionOptions("native", vendor)
```

```
opts =  
    SqlConnectionOptions with properties:  
  
        DataSourceName: ""  
        Vendor: "PostgreSQL"  
  
        DatabaseName: ""  
        Server: "localhost"  
        PortNumber: 5432
```

`opts` is an `SqlConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server

- PortNumber — Port number

Configure the data source by setting the database connection options for the data source PostgreSQLDataSource, database name toystore_doc, database server dbtb00, and port number 5432.

```
opts = setoptions(opts, ...
    'DataSourceName', "PostgreSQLDataSource", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb00", ...
    'PortNumber', 5432)
```

```
opts =
    SQLConnectionOptions with properties:
```

```
    DataSourceName: "PostgreSQLDataSource"
    Vendor: "PostgreSQL"
```

```
    DatabaseName: "toystore_doc"
    Server: "dbtb00"
    PortNumber: 5432
```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection with a user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "dbdev";
password = "matlab";
status = testConnection(opts, username, password)
```

```
status = logical
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `postgresql` function or the Database Explorer app.

Version History

Introduced in R2020b

See Also

`databaseConnectionOptions` | `deleteDataSource`

Topics

“Configure PostgreSQL Native Interface Data Source” on page 7-2

“Import Data from PostgreSQL Database Table” on page 7-11

reset

Package: `database.options.native.postgresql`

Reset PostgreSQL native interface connection options to defaults

Syntax

```
opts = reset(opts)
```

Description

`opts = reset(opts)` resets all connection options to their default values for all properties of the `SQLConnectionOptions` object. The `reset` function also removes any additional driver-specific properties from the `SQLConnectionOptions` object.

Examples

Reset Connection Options to Default Values

Create, configure, and test a PostgreSQL native interface data source for a PostgreSQL database. Reset the database connection options to their default values. Then configure, test, and save the data source with different database connection options.

Create a PostgreSQL native interface data source for a PostgreSQL database connection.

```
vendor = "PostgreSQL";  
opts = databaseConnectionOptions("native", vendor)
```

```
opts =  
  SQLConnectionOptions with properties:  
    DataSourceName: ""  
    Vendor: "PostgreSQL"  
  
    DatabaseName: ""  
    Server: "localhost"  
    PortNumber: 5432
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Configure the data source by setting the database connection options for the data source `PostgreSQLDataSource`, database name `toystore_doc`, database server `dbtb00`, and port number 5432.

```
opts = setoptions(opts, ...
    'DataSourceName','PostgreSQLDataSource', ...
    'DatabaseName','toystore_doc','Server','dbtb00', ...
    'PortNumber',5432)
```

```
opts =
    SQLConnectionOptions with properties:

        DataSourceName: "PostgreSQLDataSource"
           Vendor: "PostgreSQL"

        DatabaseName: "toystore_doc"
           Server: "dbtb00"
        PortNumber: 5432
```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection with a user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "dbdev";
password = "matlab";
status = testConnection(opts,username,password)

status = logical
        1
```

Reset the database connection options to their default values.

```
opts = reset(opts)

opts =
    SQLConnectionOptions with properties:

        DataSourceName: ""
           Vendor: "PostgreSQL"

        DatabaseName: ""
           Server: "localhost"
        PortNumber: 5432
```

Configure the data source again by setting the database connection options for the data source `PostgreSQLDataSource`, database name `toystore_doc`, database server `dbtb00`, and port number 5432. Also, set a driver-specific connection option to specify a timeout value for establishing the database connection.

```
opts = setoptions(opts, ...
    "DataSourceName","PostgreSQLDataSource", ...
    "DatabaseName","toystore_doc", ...
    "Server","dbtb00","PortNumber",5432, ...
    "connect_timeout",20)
```

```
opts =  
    SqlConnectionOptions with properties:  
  
        DataSourceName: "PostgreSQLDataSource"  
        Vendor: "PostgreSQL"  
  
        DatabaseName: "toystore_doc"  
        Server: "dbtb00"  
        PortNumber: 5432  
  
    Additional Connection Options:  
  
        connect_timeout: 20
```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SqlConnectionOptions` object. The driver-specific connection option appears below the other connection options.

Test the database connection again.

```
username = "dbdev";  
password = "matlab";  
status = testConnection(opts,username,password)  
  
status = logical  
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

`SqlConnectionOptions` object

Database connection options, specified as an `SqlConnectionOptions` object.

Output Arguments

opts — Database connection options

`SqlConnectionOptions` object

Database connection options, returned as an `SqlConnectionOptions` object.

Version History

Introduced in R2020b

See Also

Objects

SQLConnectionOptions

Functions

databaseConnectionOptions | roptions | saveAsDataSource | setoptions |
testConnection | deleteDataSource

Topics

“Import Data from PostgreSQL Database Table” on page 7-11

“Customize Options for Importing Data from PostgreSQL Database into MATLAB” on page 7-13

rmoptions

Package: `database.options.native.postgresql`

Remove PostgreSQL native interface connection options

Syntax

```
opts = rmoptions(opts,option)
```

Description

`opts = rmoptions(opts,option)` removes one or more specified connection options from the `SQLConnectionOptions` object `opts`.

Examples

Remove Driver-Specific Connection Option

Edit an existing PostgreSQL native interface data source for a PostgreSQL database. Set an additional driver-specific option, and test the database connection. Then, remove the additional driver-specific option, and test and save the data source.

Retrieve the existing PostgreSQL native interface data source.

```
datasource = "PostgreSQLDataSource";
opts = databaseConnectionOptions(datasource)

opts =
  SQLConnectionOptions with properties:

      DataSourceName: "PostgreSQLDataSource"
      Vendor: "PostgreSQL"

      DatabaseName: "toystore_doc"
      Server: "dbtb00"
      PortNumber: 5432
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Add a driver-specific connection option by using a name-value pair argument. The option specifies a timeout value for establishing the database connection. `opts` contains a new property for the additional connection option.

```

opts = setoptions(opts,"connect_timeout",20)

opts =
  SqlConnectionOptions with properties:

      DataSourceName: "PostgreSQLDataSource"
      Vendor: "PostgreSQL"

      DatabaseName: "toystore_doc"
      Server: "dbtb00"
      PortNumber: 5432

  Additional Connection Options:

      connect_timeout: 20

```

Test the database connection with a user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```

username = "dbdev";
password = "matlab";
status = testConnection(opts,username,password)

status = logical
      1

```

Remove the driver-specific option for specifying a timeout value. The `opts` object no longer contains the `connect_timeout` property.

```

opts = rmoptions(opts,"connect_timeout")

opts =
  SqlConnectionOptions with properties:

      DataSourceName: "PostgreSQLDataSource"
      Vendor: "PostgreSQL"

      DatabaseName: "toystore_doc"
      Server: "dbtb00"
      PortNumber: 5432

```

Test the database connection again.

```

username = "dbdev";
password = "matlab";
status = testConnection(opts,username,password)

status = logical
      1

```

Save the data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, specified as an SQLConnectionOptions object.

option — PostgreSQL native interface connection option

character vector | string scalar | cell array of character vectors | string array

PostgreSQL native interface connection option, specified as a character vector, string scalar, cell array of character vectors, or string array. Specify the name of one or more PostgreSQL native interface connection options or driver-specific connection options.

Example: ["DatabaseName" "Server" "PortNumber"]

Example: "connect_timeout"

Data Types: char | string | cell

Output Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, returned as an SQLConnectionOptions object.

Version History

Introduced in R2020b

See Also

Objects

SQLConnectionOptions

Functions

databaseConnectionOptions | postgresql | saveAsDataSource | setoptions | reset | testConnection | deleteDataSource

Topics

“Import Data from PostgreSQL Database Table” on page 7-11

“Customize Options for Importing Data from PostgreSQL Database into MATLAB” on page 7-13

saveAsDataSource

Package: database.options.native.postgresql

Save PostgreSQL native interface data source

Syntax

```
saveAsDataSource(opts)
```

Description

`saveAsDataSource(opts)` saves the PostgreSQL native interface data source specified by the `SQLConnectionOptions` object `opts`.

Examples

Create PostgreSQL Native Interface Data Source

Create, configure, test, and save a PostgreSQL native interface data source for a PostgreSQL database.

Create a PostgreSQL native interface data source for a PostgreSQL native interface database connection.

```
vendor = "PostgreSQL";  
opts = databaseConnectionOptions("native", vendor)
```

```
opts =  
  SQLConnectionOptions with properties:  
  
      DataSourceName: ""  
      Vendor: "PostgreSQL"  
  
      DatabaseName: ""  
      Server: "localhost"  
      PortNumber: 5432
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Configure the data source by setting the database connection options for the data source `PostgreSQLDataSource`, database name `toystore_doc`, database server `dbtb00`, and port number `5432`.

```
opts = setoptions(opts, ...  
    'DataSourceName','PostgreSQLDataSource', ...  
    'DatabaseName','toystore_doc','Server','dbtb00', ...  
    'PortNumber',5432)
```

```
opts =  
    SQLConnectionOptions with properties:  
  
        DataSourceName: "PostgreSQLDataSource"  
        Vendor: "PostgreSQL"  
  
        DatabaseName: "toystore_doc"  
        Server: "dbtb00"  
        PortNumber: 5432
```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection with a user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "dbdev";  
password = "matlab";  
status = testConnection(opts,username,password)
```

```
status = logical  
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `postgresql` function or the Database Explorer app.

Input Arguments

opts — Database connection options

`SQLConnectionOptions` object

Database connection options, specified as an `SQLConnectionOptions` object.

Version History

Introduced in R2020b

See Also

Objects

`SQLConnectionOptions`

Functions

databaseConnectionOptions | postgresql | roptions | setoptions | reset |
testConnection | deleteDataSource

Topics

“Import Data from PostgreSQL Database Table” on page 7-11

“Customize Options for Importing Data from PostgreSQL Database into MATLAB” on page 7-13

setoptions

Package: `database.options.native.postgresql`

Set PostgreSQL native interface connection options

Syntax

```
opts = setoptions(opts,Option1,OptionValue1,...,OptionN,OptionValueN)
```

Description

`opts = setoptions(opts,Option1,OptionValue1,...,OptionN,OptionValueN)` sets connection options using the `SQLConnectionOptions` object `opts`.

Examples

Create PostgreSQL Native Interface Data Source

Create, configure, test, and save a PostgreSQL native interface data source for a PostgreSQL database.

Create a PostgreSQL native interface data source for a PostgreSQL native interface database connection.

```
vendor = "PostgreSQL";  
opts = databaseConnectionOptions("native",vendor)
```

```
opts =  
  SQLConnectionOptions with properties:  
  
      DataSourceName: ""  
      Vendor: "PostgreSQL"  
  
      DatabaseName: ""  
      Server: "localhost"  
      PortNumber: 5432
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Configure the data source by setting the database connection options for the data source `PostgreSQLDataSource`, database name `toystore_doc`, database server `dbtb00`, and port number `5432`.

```
opts = setoptions(opts, ...
    'DataSourceName','PostgreSQLDataSource', ...
    'DatabaseName','toystore_doc','Server','dbtb00', ...
    'PortNumber',5432)
```

```
opts =
    SQLConnectionOptions with properties:

        DataSourceName: "PostgreSQLDataSource"
        Vendor: "PostgreSQL"

        DatabaseName: "toystore_doc"
        Server: "dbtb00"
        PortNumber: 5432
```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection with a user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "dbdev";
password = "matlab";
status = testConnection(opts,username,password)

status = logical
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `postgresql` function or the Database Explorer app.

Edit Existing PostgreSQL Native Interface Data Source

Edit an existing PostgreSQL native interface data source for a PostgreSQL database. Set an additional driver-specific option and save the data source.

Retrieve the existing PostgreSQL data source `PostgreSQLDataSource`.

```
datasource = "PostgreSQLDataSource";
opts = databaseConnectionOptions(datasource)

opts =
    SQLConnectionOptions with properties:

        DataSourceName: "PostgreSQLDataSource"
        Vendor: "PostgreSQL"

        DatabaseName: "toystore_doc"
        Server: "dbtb00"
```

```
PortNumber: 5432
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Add a driver-specific connection option by using a name-value pair argument. The option specifies a timeout value for establishing the database connection. `opts` contains a new section of properties for the additional connection option.

```
opts = setoptions(opts,"connect_timeout",20)
```

```
opts =  
    SQLConnectionOptions with properties:  
  
        DataSourceName: "PostgreSQLDataSource"  
        Vendor: "PostgreSQL"  
  
        DatabaseName: "toystore_doc"  
        Server: "dbtb00"  
        PortNumber: 5432  
  
    Additional Connection Options:  
  
        connect_timeout: 20
```

Test the database connection with a user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "dbdev";  
password = "matlab";  
status = testConnection(opts,username,password)  
  
status = logical  
        1
```

Save the updated data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

`SQLConnectionOptions` object

Database connection options, specified as an `SQLConnectionOptions` object.

Option1,OptionValue1,...,OptionN,OptionValueN — PostgreSQL native interface connection options

name-value pair arguments

PostgreSQL native interface connection options, specified as one or more name-value pair arguments. **Option** is a character vector or string scalar that specifies the name of a PostgreSQL native interface connection option. **OptionValue** specifies the value of the option. **OptionValue** can be a character vector, string scalar, logical scalar, or numeric scalar. You can specify any PostgreSQL native interface connection option that is a property of the `SQLConnectionOptions` object.

Example: `'DataSourceName','myDataSource','Server','localhost','PortNumber',5432` configures a PostgreSQL native interface data source named `myDataSource` that is located on the local server with the port number 5432.

Output Arguments

opts — Database connection options

`SQLConnectionOptions` object

Database connection options, returned as an `SQLConnectionOptions` object.

Version History

Introduced in R2020b

See Also

Objects

`SQLConnectionOptions`

Functions

`databaseConnectionOptions` | `postgresql` | `roptions` | `reset` | `saveAsDataSource` | `testConnection` | `deleteDataSource`

Topics

“Import Data from PostgreSQL Database Table” on page 7-11

“Customize Options for Importing Data from PostgreSQL Database into MATLAB” on page 7-13

testConnection

Package: database.options.native.postgresql

Test PostgreSQL native interface database connection

Syntax

```
status = testConnection(opts,username,password)
[status,message] = testConnection(opts,username,password)
```

Description

`status = testConnection(opts,username,password)` tests the PostgreSQL native interface database connection specified by the `SQLConnectionOptions` object `opts`, a user name, and a password.

`[status,message] = testConnection(opts,username,password)` also returns the error message associated with testing the database connection.

Examples

Create PostgreSQL Native Interface Data Source

Create, configure, test, and save a PostgreSQL native interface data source for a PostgreSQL database.

Create a PostgreSQL native interface data source for a PostgreSQL native interface database connection.

```
vendor = "PostgreSQL";
opts = databaseConnectionOptions("native",vendor)
```

```
opts =
  SQLConnectionOptions with properties:
      DataSourceName: ""
      Vendor: "PostgreSQL"
      DatabaseName: ""
      Server: "localhost"
      PortNumber: 5432
```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server

- PortNumber — Port number

Configure the data source by setting the database connection options for the data source `PostgreSQLDataSource`, database name `toystore_doc`, database server `dbtb00`, and port number `5432`.

```
opts = setoptions(opts, ...
    'DataSourceName','PostgreSQLDataSource', ...
    'DatabaseName','toystore_doc','Server','dbtb00', ...
    'PortNumber',5432)
```

```
opts =
    SQLConnectionOptions with properties:
```

```
    DataSourceName: "PostgreSQLDataSource"
    Vendor: "PostgreSQL"
```

```
    DatabaseName: "toystore_doc"
    Server: "dbtb00"
    PortNumber: 5432
```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection with a user name and password. The `testConnection` function returns the logical `1`, which indicates the database connection is successful.

```
username = "dbdev";
password = "matlab";
status = testConnection(opts,username,password)
```

```
status = logical
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

You can connect to the new data source using the `postgresql` function or the Database Explorer app.

Retrieve Message for PostgreSQL Native Interface Database Connection Test

Create and configure a PostgreSQL native interface data source to a PostgreSQL database. Test the database connection to the PostgreSQL native interface data source and retrieve the error message.

Create a PostgreSQL native interface data source for a PostgreSQL database connection.

```
vendor = "PostgreSQL";
opts = databaseConnectionOptions("native",vendor)
```

```
opts =
    SQLConnectionOptions with properties:
```

```

DataSourceName: ""
    Vendor: "PostgreSQL"

DatabaseName: ""
    Server: "localhost"
    PortNumber: 5432

```

`opts` is an `SQLConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `DatabaseName` — Name of the database
- `Server` — Name of the database server
- `PortNumber` — Port number

Configure the data source by setting the database connection options for the data source `PostgreSQLDataSource`, database name `toystore_doc`, database server `dbtb00`, and port number 5432.

```

opts = setoptions(opts, ...
    'DataSourceName', "PostgreSQLDataSource", ...
    'DatabaseName', "toystore_doc", 'Server', "dbtb00", ...
    'PortNumber', 5432)

```

`opts` =
`SQLConnectionOptions` with properties:

```

DataSourceName: "PostgreSQLDataSource"
    Vendor: "PostgreSQL"

DatabaseName: "toystore_doc"
    Server: "dbtb00"
    PortNumber: 5432

```

The `setoptions` function sets the `DataSourceName`, `DatabaseName`, `Server`, and `PortNumber` properties in the `SQLConnectionOptions` object.

Test the database connection using an incorrect user name and password. The `testConnection` function returns the logical 0, which indicates the database connection fails. Retrieve and display the error message for the failed connection.

```

username = "wronguser";
password = "wrongpassword";
[status,message] = testConnection(opts,username,password)

status = logical
    0

message =
    'Driver Error: FATAL: password authentication failed for user "wronguser"

```



```
FATAL: password authentication failed for user "wronguser"
```

Input Arguments

opts — Database connection options

SQLConnectionOptions object

Database connection options, specified as an SQLConnectionOptions object.

username — User name

character vector | string scalar

User name required to access the database, specified as a character vector or string scalar. If no user name is required, specify an empty value "".

Data Types: char | string

password — Password

character vector | string scalar

Password required to access the database, specified as a character vector or string scalar. If no password is required, specify an empty value "".

Data Types: char | string

Output Arguments

status — Connection status

logical

Connection status, returned as a logical `true` if the connection test passes or `false` if the connection test fails.

message — Error message

character vector

Error message, returned as a character vector. If the connection test passes, then the error message is an empty character vector. Otherwise, the error message contains text describing the failed database connection.

Version History

Introduced in R2020b

See Also

Objects

SQLConnectionOptions

Functions

databaseConnectionOptions | postgresql | roptions | reset | setoptions | saveAsDataSource | deleteDataSource

Topics

“Import Data from PostgreSQL Database Table” on page 7-11

“Customize Options for Importing Data from PostgreSQL Database into MATLAB” on page 7-13

execute

Package: database.postgre

Execute SQL statement using PostgreSQL native interface database connection

Syntax

```
execute(conn,sqlquery)
```

Description

`execute(conn,sqlquery)` executes an SQL query that contains a non-SELECT SQL statement by using the relational database connection.

Examples

Execute Non-SELECT SQL Statement

Using the PostgreSQL native interface, create and execute a non-SELECT SQL statement that deletes a database table. The `PostgreSQLDataSource` data source configures a database connection to a PostgreSQL database.

This example uses a PostgreSQL database version 9.405 database and the libpq driver version 10.12.

Connect to the database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
```

```
conn = postgresql(datasource,username,password);
```

Load patient information into the MATLAB® workspace.

```
patients = readtable("patients.xls");
```

Create the `patients` database table using the patient information.

```
tablename = "patients";
sqlwrite(conn,tablename,patients)
```

Import the data from the `patients` database table.

```
data = sqlread(conn,tablename);
```

Delete the `patients` database table using the `execute` function.

```
sqlquery = strcat("DROP TABLE ",tablename);
execute(conn,sqlquery)
```

Ensure that the table no longer exists.

```
data = sqlfind(conn,tablename)
```

```
data =
```

```
    0x5 empty table
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar. The SQL statement can be any valid non-SELECT SQL statement.

The SQL statement can be a stored procedure that does not return any result sets. For stored procedures that return one or more result sets, use the `fetch` function.

For information about the SQL query language, see the PostgreSQL Documentation.

Example: "DROP TABLE patients"

Data Types: char | string

Version History

Introduced in R2020b

See Also

postgresql | close | fetch | sqlread

Topics

"Create Table and Add Column Using PostgreSQL Native Interface" on page 7-23

"Delete Data from Database Using PostgreSQL Native Interface" on page 7-24

External Websites

PostgreSQL Documentation

executeSQLScript

Package: database.postgre

Execute SQL script on PostgreSQL database

Syntax

```
results = executeSQLScript(conn,scriptfile)
results = executeSQLScript(conn,scriptfile,Name,Value)
```

Description

`results = executeSQLScript(conn,scriptfile)` uses the database connection `conn` to return a structure array that contains results as a table (by default) for each executed SQL SELECT statement in the SQL script file. For any non-SELECT SQL statements, the corresponding table is empty. The `executeSQLScript` function executes all SQL statements in the SQL script file.

`results = executeSQLScript(conn,scriptfile,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'DataReturnFormat','cellarray'` stores the results of an executed SQL statement as a cell array. The results are stored in the `Data` field of the `results` structure array.

Examples

Execute SQL Script Using PostgreSQL Native Interface

Connect to a PostgreSQL database. Then, run two SQL SELECT statements from the SQL script file `compare_sales.sql`, import the results, and perform simple sales data analysis. The file contains two SQL queries: the first retrieves sales of products from US suppliers, and the second retrieves sales of products from foreign suppliers.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Execute the SQL script. The SQL script contains two SQL queries that retrieve sales data from US and foreign suppliers, respectively.

```
scriptfile = "compare_sales.sql";
results = executeSQLScript(conn,scriptfile)
```

```
results=1x2 struct array with fields:
    SQLQuery
    Data
```

Message

The `executeSQLScript` function returns a structure array that contains two tables in the `Data` field. The first table contains the results of executing the first SQL query in the SQL script file. The second table contains the results of executing the second SQL query.

Display the first eight rows of imported data for the second SQL query in the SQL script file. The data shows sales results from foreign suppliers.

```
data = head(results(2).Data)
```

```
data=4x6 table
  productdescription      suppliername      city      jan_sales      feb_sales      mar_sales
  _____      _____      _____      _____      _____      _____
  "Victorian Doll"      "Wacky Widgets"      "Adelaide"      1400      1100      9800
  "Painting Set"      "Terrific Toys"      "London"      3000      2400      18000
  "Sail Boat"      "Incredible Machines"      "Dublin"      3000      2400      15000
  "Slinky"      "Doll's Galore"      "London"      3000      1500      10000
```

Retrieve the variable names in the table.

```
names = data.Properties.VariableNames
```

```
names = 1x6 cell
  {'productdescription'}      {'suppliername'}      {'city'}      {'jan_sales'}      {'feb_sales'}
```

Determine the highest sales amount in January.

```
max(data.jan_sales)
```

```
ans = 3000
```

Close the database connection.

```
close(conn)
```

Execute SQL Script and Return Results as Structures

Connect to a PostgreSQL database. Then, run two SQL `SELECT` statements from the SQL script file `compare_sales.sql`. Import the results from the SQL queries as structures and perform simple sales data analysis. The file contains two SQL queries: the first retrieves sales of products from US suppliers, and the second retrieves sales of products from foreign suppliers.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Execute the SQL script. The SQL script contains two SQL queries that retrieve sales data from US and foreign suppliers, respectively. Specify `structure` as the data return format for the query results.

```
scriptfile = "compare_sales.sql";
results = executeSQLScript(conn,scriptfile, ...
    'DataReturnFormat','structure')

results=1x2 struct array with fields:
    SQLQuery
    Data
    Message
```

The `executeSQLScript` function returns a structure array that contains two structures in the `Data` field. The first structure contains the results of executing the first SQL query in the SQL script file. The second structure contains the results of executing the second SQL query.

Display the imported data for the second SQL query in the SQL script file. The data contains sales results from foreign suppliers.

```
data = results(2).Data

data=4x1 struct array with fields:
    productdescription
    suppliername
    city
    jan_sales
    feb_sales
    mar_sales
```

Determine the highest sales amount in January.

```
for i = 1:length(data)
    jan_sales(i,1) = data(i).jan_sales;
end
max(jan_sales)

ans = 3000
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

scriptfile — Name of SQL script file

character vector | string scalar

Name of SQL script file that contains one or more SQL statements to run, specified as a character vector or string scalar. The file must be a text file and can contain comments in addition to SQL queries. Start single-line comments with `--`. Enclose multiline comments in `/*...*/`.

The SQL script file can contain one or more SQL statements terminated by either a semicolon or the keyword `GO`. The following is an example of two SQL `SELECT` statements.

```
SELECT productdescription, suppliername
FROM suppliers a, producttable b
WHERE a.suppliernumber = b.suppliernumber;
```

```
SELECT suppliername, country
FROM suppliers;
```

Example: `'C:\work\sql_file.sql'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `results = executeSQLScript(conn, scriptfile, 'DataReturnFormat', "numeric", 'ErrorHandling', "store")` returns query results as a numeric matrix in the `Data` field of the `results` structure array and stores any error message in the `Message` field of `results`.

DataReturnFormat — Data return format

`"table"` (default) | `"cellarray"` | `"numeric"` | `"structure"`

Data return format, specified as the comma-separated pair consisting of `'DataReturnFormat'` and one of these values:

- `"table"`
- `"cellarray"`
- `"numeric"`
- `"structure"`

You can specify the value using a character vector or string scalar.

The `'DataReturnFormat'` name-value pair argument specifies the data type of the `Data` field in the `results` structure array.

Example: `'DataReturnFormat', "structure"` returns a structure array that contains query results stored in structures.

ErrorHandling — Error handling

`"report"` (default) | `"store"`

Error handling, specified as the comma-separated pair consisting of `'ErrorHandling'` and one of these values:

- "report" — When an SQL statement fails to execute, stop execution of the remaining SQL statements in the SQL script file and display an error message at the command line.
- "store" — When an SQL statement fails to execute, store an error message in the Message field of the results structure array.

You can specify the value using a character vector or string scalar.

Example: 'ErrorHandling', "report" displays an error message at the command line.

Output Arguments

results — Query results

structure array

Query results from executed SQL statements in the SQL script file, returned as a structure array with these fields.

Field Name	Field Data Type	Field Description
SQLQuery	character vector	Stores the SQL statement or statements executed in the SQL script file.
Data	<ul style="list-style-type: none"> • table (default) • cell array • numeric matrix • structure 	<p>Stores the results of executed SQL SELECT statements.</p> <p>The 'DataReturnFormat' name-value pair argument specifies the data type of the Data field.</p> <p>For non-SELECT SQL statements, the Data field is an empty table, which means the executed SQL query has no results.</p>
Message	character vector	<p>Stores an error message for the respective SQL statement that fails to execute.</p> <p>The Message field contains an error message only if you specify the 'ErrorHandling' name-value pair argument with the value "store".</p>

The number of elements in the structure array is equal to the number of SQL statements in the SQL script file. `results(M)` contains the results from executing the Mth SQL statement in the SQL script file. If the SQL statement returns query results, then the results are stored in `results(M).Data`.

For details about accessing structure arrays, see "Structure Arrays".

Limitations

- Use the `executeSQLScript` function to import data into MATLAB, especially if you have long and complex SQL queries that are difficult to convert into MATLAB character vectors or string scalars. The `executeSQLScript` function does not support SQL scripts containing continuous PL/SQL blocks with BEGIN and END, such as stored procedure definitions or trigger definitions. However, `executeSQLScript` does support table definitions.

- An SQL script containing either of the following can produce unexpected results:
 - Apostrophes that are not escaped, including those in comments. For example, write the character vector 'Here's the code' as 'Here''s the code'.
 - Nested comments.
- An SQL script containing more than 25,000 characters causes the `executeSQLScript` function to return an error.

Version History

Introduced in R2020b

See Also

`postgresql` | `close` | `commit` | `rollback`

Topics

“Import Data from PostgreSQL Database Table” on page 7-11

“Generate SQL Query and MATLAB Script” on page 4-20

External Websites

PostgreSQL Documentation

fetch

Package: database.postgre

Import results of SQL statement in PostgreSQL database into MATLAB

Syntax

```
results = fetch(conn,sqlquery)
results = fetch(conn,sqlquery,opts)
results = fetch( ___,Name,Value)
[results,metadata] = fetch( ___ )
```

Description

`results = fetch(conn,sqlquery)` returns all rows of data after executing the SQL statement `sqlquery` for the connection object. `fetch` imports data in batches.

`results = fetch(conn,sqlquery,opts)` customizes options for importing data from an executed SQL query by using the `SQLImportOptions` object.

`results = fetch(___,Name,Value)` specifies additional options using one or more name-value arguments with any of the previous input argument combinations. For example, specify `MaxRows = 5` to import five rows of data.

`[results,metadata] = fetch(___)` also returns the `metadata` table, which contains metadata information about the imported data.

Examples

Import All Data Using PostgreSQL Native Interface

Import all product data from a PostgreSQL database table into MATLAB® using the PostgreSQL native interface and the `fetch` function. Determine the highest unit cost among products in the table. Then, use a row filter to import only the data for products with a unit cost less than 15.

Create a PostgreSQL native interface database connection to a PostgreSQL database using a data source, username, and password. The database contains the table `productTable`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Import all the data from `productTable` by using the connection object and SQL query. Then, display the first five rows of the imported data.

```
sqlquery = "SELECT * FROM productTable";
data = fetch(conn,sqlquery);
head(data,5)
```

productnumber	stocknumber	suppliernumber	unitcost	productdescription
9	1.2597e+05	1003	13	"Victorian Doll"
8	2.1257e+05	1001	5	"Train Set"
7	3.8912e+05	1007	16	"Engine Kit"
2	4.0031e+05	1002	9	"Painting Set"
4	4.0034e+05	1008	21	"Space Cruiser"

Determine the highest unit cost for all products in the table.

```
max(data.unitcost)
```

```
ans = 24
```

Now, import the data using a row filter. The filter condition is that `unitcost` must be less than 15.

```
rf = rowfilter("unitcost");
rf = rf.unitcost < 15;
data = fetch(conn,sqlquery,"RowFilter",rf);
```

Again, display the first five rows of the imported data.

```
head(data,5)
```

productnumber	stocknumber	suppliernumber	unitcost	productdescription
9	1.2597e+05	1003	13	"Victorian Doll"
8	2.1257e+05	1001	5	"Train Set"
2	4.0031e+05	1002	9	"Painting Set"
1	4.0034e+05	1001	14	"Building Blocks"
5	4.0046e+05	1005	3	"Tin Soldier"

Close the database connection.

```
close(conn)
```

Import Data from SQL Query Using Import Options

Customize import options when importing data from the results of an SQL query on a PostgreSQL database using the PostgreSQL native interface. Control the import options by creating an `SQLImportOptions` object. Then, customize import options for different columns in the SQL query. Import data using the `fetch` function.

This example uses the `employees_database.mat` file, which contains the columns `first_name`, `hire_date`, and `department_name`. The example uses a PostgreSQL database version 9.405 database and the `libpq` driver version 10.12.

Create a PostgreSQL native interface database connection to a PostgreSQL database with a data source name, username, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
```

```
conn = postgresql(datasource,username,password);
```

Load employee information into the MATLAB® workspace.

```
employeedata = load("employees_database.mat");
```

Create the employees and departments database tables using the employee information.

```
emps = employeedata.employees;
depts = employeedata.departments;
```

```
sqlwrite(conn,"employees",emps)
sqlwrite(conn,"departments",depts)
```

Create an SQLImportOptions object using an SQL query and the databaseImportOptions function. This query retrieves all information for employees who are sales managers or programmers.

```
sqlquery = strcat("SELECT * from employees e join departments d ", ...
    "on (e.department_id = d.department_id) WHERE ", ...
    "(job_id = 'IT_PROG' or job_id = 'SA_MAN')");
opts = databaseImportOptions(conn,sqlquery)
```

```
opts =
```

```
SQLImportOptions with properties:
```

```
    ExcludeDuplicates: false
    VariableNamingRule: 'preserve'

    VariableNames: {'employee_id', 'first_name', 'last_name' ... and 13 more}
    VariableTypes: {'double', 'string', 'string' ... and 13 more}
    SelectedVariableNames: {'employee_id', 'first_name', 'last_name' ... and 13 more}
    FillValues: { NaN, <missing>, <missing> ... and 13 more }
    RowFilter: <unconstrained>

    VariableOptions: Show all 16 VariableOptions
```

Display the current import options for the variables selected in the SelectedVariableNames property of the SQLImportOptions object.

```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)
```

```
varOpts =
```

```
1x16 SQLVariableImportOptions array with properties:
```

```
Variable Options:
```

	(1)	(2)	(3)	(4)	(5)	
Name:	'employee_id'	'first_name'	'last_name'	'email'	'phone_number'	'hire_date'
Type:	'double'	'string'	'string'	'string'	'string'	'datetime'
MissingRule:	'fill'	'fill'	'fill'	'fill'	'fill'	'fill'
FillValue:	NaN	<missing>	<missing>	<missing>	<missing>	<missing>

To access sub-properties of each variable, use getoptions

Change the data types for the hire_date, department_name, and first_name variables using the setoptions function. Then, display the updated import options. For efficiency, change the data type

of the `hire_date` variable to `string`. Because `department_name` designates a finite set of repeating values, change the data type of this variable to `categorical`. Because `first_name` stores text data, change the data type of this variable to `char`.

```
opts = setoptions(opts,"hire_date","Type","string");
opts = setoptions(opts,"department_name","Type","categorical");
opts = setoptions(opts,"first_name","Type","char");
```

```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)
```

```
varOpts =
  1x16 SQLVariableImportOptions array with properties:
```

Variable Options:

	(1)	(2)	(3)	(4)	(5)	
Name:	'employee_id'	'first_name'	'last_name'	'email'	'phone_number'	'hire_date'
Type:	'double'	'char'	'string'	'string'	'string'	'string'
MissingRule:	'fill'	'fill'	'fill'	'fill'	'fill'	'fill'
FillValue:	NaN	' '	<missing>	<missing>	<missing>	<missing>

To access sub-properties of each variable, use `getoptions`

Select the three modified variables using the `SelectVariableNames` property.

```
opts.SelectedVariableNames = ["first_name","hire_date","department_name"];
```

Set the filter condition to import only the data for the employees hired before January 1, 2006.

```
opts.RowFilter = opts.RowFilter.hire_date < datetime(2006,01,01)
```

```
opts =
  SQLImportOptions with properties:
```

```
ExcludeDuplicates: false
VariableNamingRule: 'preserve'

VariableNames: {'employee_id', 'first_name', 'last_name' ... and 13 more}
VariableTypes: {'double', 'char', 'string' ... and 13 more}
SelectedVariableNames: {'first_name', 'hire_date', 'department_name'}
FillValues: { NaN, ' ', <missing> ... and 13 more }
RowFilter: hire_date < 01-Jan-2006
```

VariableOptions: Show all 16 VariableOptions

Import and display the results of the SQL query using the `fetch` function.

```
employees_data = fetch(conn,sqlquery,opts)
```

```
employees_data=4x3 table
  first_name      hire_date      department_name
  _____      _____      _____
  {'David' }      "2005-06-25 00:00:00"      IT
  {'Alberto'}     "2005-03-10 00:00:00"      Sales
  {'Karen' }      "2005-01-05 00:00:00"      Sales
  {'John' }       "2004-10-01 00:00:00"      Sales
```

Delete the employees and departments database tables using the execute function.

```
execute(conn, "DROP TABLE employees")
execute(conn, "DROP TABLE departments")
```

Close the database connection.

```
close(conn)
```

Import Data from SQL Query as Structure

Specify the data return format and the number of imported rows for the results of an SQL query. Import data using the SQL query and the fetch function.

This example uses a PostgreSQL database version 9.4.05 database and the libpq driver version 10.12.

Create a PostgreSQL native interface database connection to a PostgreSQL database with a data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
```

```
conn = postgresql(datasource, username, password);
```

Load patient information into the MATLAB® workspace.

```
patients = readtable('patients.xls');
```

Create the patients database table using the patient information.

```
tablename = "patients";
sqlwrite(conn, tablename, patients)
```

Select all data from the patients database table and import five rows from the table as a structure. Use the 'DataReturnFormat' name-value pair argument to specify returning the data as a structure. Also, use the 'MaxRows' name-value pair argument to specify five rows. Display the imported data.

```
sqlquery = strcat("SELECT * FROM ", tablename);
results = fetch(conn, sqlquery, 'DataReturnFormat', "structure", ...
    'MaxRows', 5)
```

results=5x1 struct array with fields:

```
    lastname
    gender
    age
    location
    height
    weight
    smoker
    systolic
    diastolic
    selfassessedhealthstatus
```

Delete the `patients` database table using the `execute` function.

```
sqlquery = strcat("DROP TABLE ",tablename);  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Retrieve Metadata Information About Imported Data

Retrieve metadata information when importing data from an SQL query. Import data using the `fetch` function and explore the metadata information by using dot notation.

This example uses the `outages.csv` file, which contains outage data. Also, the example uses a PostgreSQL database version 9.405 database and the `libpq` driver version 10.12.

Create a PostgreSQL native interface database connection to a PostgreSQL database with a data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";  
username = "dbdev";  
password = "matlab";  
  
conn = postgresql(datasource,username,password);
```

Load outage information into the MATLAB® workspace.

```
outages = readtable("outages.csv");
```

Create the `outages` database table using the outage information. Use the `'ColumnType'` name-value pair argument to customize the data types of the variables in the `outages` table.

```
tablename = "outages";  
sqlwrite(conn,tablename,outages, ...  
    'ColumnType',[ "varchar(120)", "timestamp", "numeric(38,16)", ...  
    "numeric(38,16)", "timestamp", "varchar(150)"])
```

Import the data into the MATLAB workspace and return metadata information about the imported data.

```
sqlquery = "SELECT * FROM outages";  
[results,metadata] = fetch(conn,sqlquery);
```

View the names of the variables in the imported data.

```
metadata.Properties.RowNames
```

```
ans = 6x1 cell  
    {'region'      }  
    {'outagetime'  }  
    {'loss'        }  
    {'customers'   }  
    {'restorationtime'}  
    {'cause'       }
```


View the data type of each variable in the imported data.

```
metadata.VariableType
```

```
ans = 6x1 cell
    {'string' }
    {'datetime'}
    {'double' }
    {'double' }
    {'datetime'}
    {'string' }
```

View the missing data value for each variable in the imported data.

```
metadata.FillValue
```

```
ans=6x1 cell array
    {1x1 missing}
    {[NaT      ]}
    {[      NaN]}
    {[      NaN]}
    {[NaT      ]}
    {1x1 missing}
```

View the indices of the missing data for each variable in the imported data.

```
metadata.MissingRows
```

```
ans=6x1 cell array
    { 0x1 double}
    { 0x1 double}
    {604x1 double}
    {328x1 double}
    { 29x1 double}
    { 0x1 double}
```

Display the first eight rows of the imported data that contain missing restoration time values. `data` contains restoration time values in the fifth variable. Use the numeric indices to find the rows with missing data.

```
index = metadata.MissingRows{5,1};
nullrestoration = results(index,:);
head(nullrestoration)
```

```
ans=8x6 table
    region      outagetime      loss      customers      restorationtime      cause
    _____  _____  _____  _____  _____  _____
    "SouthEast"  23-Jan-2003 00:49:00  530.14  2.1204e+05  NaT  "winter st
    "NorthEast"  18-Sep-2004 05:54:00      0      0  NaT  "equipment
    "MidWest"    20-Apr-2002 16:46:00  23141  NaN  NaT  "unknown"
    "NorthEast"  16-Sep-2004 19:42:00  4718  NaN  NaT  "unknown"
    "SouthEast"  14-Sep-2005 15:45:00  1839.2  3.4144e+05  NaT  "severe st
    "SouthEast"  17-Aug-2004 17:34:00  624.1  1.7879e+05  NaT  "severe st
    "SouthEast"  28-Jan-2006 23:13:00  498.78  NaN  NaT  "energy em
```

"West"	20-Jun-2003 18:22:00	0	0	NaN	"energy em
--------	----------------------	---	---	-----	------------

Delete the outages database table using the `execute` function.

```
sqlstr = "DROP TABLE ";
sqlquery = strcat(sqlstr,tablename);
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

sqlquery — SQL statement

character vector | string scalar

SQL statement, specified as a character vector or string scalar. The SQL statement can be any valid SQL statement, including nested queries. The SQL statement can be a stored procedure, such as `{call sp_name (parm1,parm2,...)}`. For stored procedures that return one or more result sets, use the `fetch` function.

Data Types: `char` | `string`

opts — Database import options

SQLImportOptions object

Database import options, specified as an SQLImportOptions object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `results =`

```
fetch(conn,sqlquery,'MaxRows',50,'DataReturnFormat','structure')
```

imports 50 rows of data as a structure.

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of `'MaxRows'` and a positive numeric scalar. By default, the `fetch` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: `'MaxRows',10`

Data Types: double

DataReturnFormat — Data return format

'table' (default) | 'cellarray' | 'numeric' | 'structure'

Data return format, specified as the comma-separated pair consisting of 'DataReturnFormat' and one of these values:

- 'table'
- 'cellarray'
- 'numeric'
- 'structure'

Use the 'DataReturnFormat' name-value pair argument to specify the data type of the results data. To specify integer classes for numeric data, use the `opts` input argument.

You can specify the value using a character vector or string scalar.

Example: 'DataReturnFormat', 'cellarray' imports data as a cell array.

VariableNamingRule — Variable naming rule

"preserve" (default) | "modify"

Variable naming rule, specified as the comma-separated pair consisting of 'VariableNamingRule' and one of these values:

- "preserve" — Preserve most variable names when the `fetch` function imports data. For details, see the [Limitations](#) section.
- "modify" — Remove non-ASCII characters from variable names when the `fetch` function imports data.

Example: 'VariableNamingRule', "modify"

Data Types: string

RowFilter — Row filter condition

<unconstrained> (default) | `matlab.io.RowFilter` object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; fetch(conn,sqlquery,"RowFilter",rf)`

Output Arguments

results — Result data

table (default) | cell array | structure | numeric matrix

Result data, returned as a table, cell array, structure, or numeric matrix. The result data contains all rows of data from the executed SQL statement by default.

Use the 'MaxRows' name-value pair argument to specify the number of rows of data to import. Use the 'DataReturnFormat' name-value pair argument to specify the data type of the result data.

When the executed SQL statement does not return any rows, the result data is an empty table.

When you import data, the `fetch` function converts the data type of each column from the PostgreSQL database to the MATLAB data type. This table maps the data type of a database column to the converted MATLAB data type.

PostgreSQL Data Type	MATLAB Data Type
Boolean	logical
Smallint	double
Integer	double
Bigint	double
Decimal	double
Numeric	double
Real	double
Double precision	double
Smallserial	double
Serial	double
Bigserial	double
Money	double
Varchar	string
Char	string
Text	string
Bytea	string
Timestamp	datetime
Timestampz	datetime
Abstime	datetime
Date	datetime
Time	duration
Timez	duration
Interval	calendarDuration
Reltime	calendarDuration
Enum	categorical
Cidr	string
Inet	string
Macaddr	string
Uuid	string
Xml	string

metadata – Metadata information

table

Metadata information, returned as a table with these variables.

Variable Name	Variable Description	Variable Data Type
VariableType	Data type of each variable in the imported data	Cell array of character vectors
FillValue	Value of missing data for each variable in the imported data	Cell array of missing data values
MissingRows	Indices for each occurrence of missing data in each variable of the imported data	Cell array of numeric indices

By default, the `fetch` function imports text data as a character vector and numeric data as a double. `FillValue` is an empty character array (for text data) or `NaN` (for numeric data) by default. To change the missing data value to another value, use the `SQLImportOptions` object.

The `RowNames` property of the metadata table contains the names of the variables in the imported data.

Limitations

The name-value argument `VariableNamingRule` has these limitations:

- The `fetch` function returns an error if you specify the `VariableNamingRule` name-value argument and set the `DataReturnFormat` name-value argument to "cellarray", "structure", or "numeric".
- The `fetch` function returns a warning if you set the `VariableNamingRule` property of the `SQLImportOptions` object to "preserve" and set the `DataReturnFormat` name-value argument to "structure".
- The `fetch` function returns an error if you use the `VariableNamingRule` name-value argument with the `SQLImportOptions` object `opts`.
- If you set the `VariableNamingRule` name-value argument to the value "modify":
 - These variable names are reserved identifiers for the `table` data type: `Properties`, `RowNames`, and `VariableNames`.
 - The length of each variable name must be less than the number returned by `namelengthmax`.

The name-value argument `RowFilter` has this limitation:

- The `fetch` function returns an error if you specify the `RowFilter` name-value argument with the `SQLImportOptions` object `opts`. It is ambiguous which of the `RowFilter` object to use in this case, especially if the filter conditions are different.

Alternative Functionality

App

The `fetch` function imports data using the command line. To import data interactively, use the **Database Explorer** app.

Version History

Introduced in R2020b

R2023a: Selectively import rows of data based on filter condition

You can use the `RowFilter` name-value argument to selectively import rows of data from a database table.

See Also**Functions**

`postgresql` | `close` | `databaseImportOptions` | `getoptions` | `reset` | `setoptions` | `execute`

Topics

“Import Data from PostgreSQL Database Table” on page 7-11

“Customize Options for Importing Data from PostgreSQL Database into MATLAB” on page 7-13

“Import Large PostgreSQL Data Using DatabaseDatastore Object” on page 7-16

External Websites

PostgreSQL Documentation

isopen

Package: database.postgre

Determine if PostgreSQL native interface database connection is open

Syntax

```
i = isopen(conn)
```

Description

`i = isopen(conn)` returns 1 if the PostgreSQL native interface database connection is open and 0 if it is closed or invalid.

Examples

Determine If Database Connection Is Open

Connect to a PostgreSQL database using the PostgreSQL native interface and verify the database connection. Then, import data from the database into MATLAB® using the database table `productTable`. Determine the highest unit cost among the retrieved products in the table. Close the database connection and ensure that the connection is closed.

Create a PostgreSQL native interface database connection using a data source, user name, and password. The PostgreSQL database contains the table `productTable`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Determine if the database connection is open. The `isopen` function returns the logical 1, which means the database connection is open.

```
i = isopen(conn)
```

```
i = logical
     1
```

Select all the data from `productTable` and sort it by the product number. `data` is a table containing the imported data that results from executing the SQL `SELECT` statement.

```
sqlquery = "SELECT * FROM productTable ORDER BY productNumber";
data = fetch(conn,sqlquery);
```

Display the first three rows of data.

```
head(data,3)
```

```
ans=3x5 table
    productnumber    stocknumber    suppliernumber    unitcost    productdescription
```

1	4.0035e+05	1001	14	"Building Blocks"
2	4.0031e+05	1002	9	"Painting Set"
3	4.01e+05	1009	17	"Slinky"

Determine the highest unit cost in the table.

```
max(data.unitcost)
```

```
ans = 24
```

Close the database connection.

```
close(conn)
```

Determine if the database connection is closed. The `isopen` function returns the logical `0`, which means the database connection is closed. If the database connection is invalid, the `isopen` function returns the same result.

```
i = isopen(conn)
```

```
i = logical  
0
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

Version History

Introduced in R2020b

See Also

`fetch` | `postgresql` | `close`

Topics

“Configure PostgreSQL Native Interface Data Source” on page 7-2

“Import Data from PostgreSQL Database Table” on page 7-11

External Websites

PostgreSQL Documentation

rollback

Package: database.postgre

Undo changes to PostgreSQL database

Syntax

```
rollback(conn)
```

Description

`rollback(conn)` reverses changes made to a database using functions such as `sqlwrite`. The `rollback` function reverses all changes made since the last `COMMIT` or `ROLLBACK` operation. To use this function, you must set the `AutoCommit` property of the connection object to `off`.

Examples

Reverse Changes Made to PostgreSQL Database

Use a PostgreSQL native interface database connection to insert product data from MATLAB® into a new table in a PostgreSQL database. Then, reverse the changes made to the database.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Allow manual committing of changes to the database by setting the `AutoCommit` property to `off`.

```
conn.AutoCommit = "off";
```

Create a MATLAB table that contains data for two products. The data is stored in the `productTable` and `suppliers` tables.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"],'VariableNames',["productNumber" ...
    "stockNumber" "supplierNumber" "unitCost" "productDescription"]);
```

Insert the product data into a new table named `toytable`.

```
tablename = "toytable";
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
  productnumber  stocknumber  suppliernumber  unitcost  productdescription
-----
           30          5e+05          1000           25      "Rubik's Cube"
           40          6e+05          2000           30      "Doll House"
```

Reverse the changes made to the database.

```
rollback(conn)
```

Search for the table. The table no longer exists.

```
data = sqlfind(conn,tablename)
```

```
data =
```

```
  0x5 empty table
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

Version History

Introduced in R2020b

See Also

postgresql | sqlwrite | commit | execute

Topics

“Insert Data into Database Table Using PostgreSQL Native Interface” on page 7-19

“Create Table and Add Column Using PostgreSQL Native Interface” on page 7-23

“Delete Data from Database Using PostgreSQL Native Interface” on page 7-24

“Roll Back Data in Database Using PostgreSQL Native Interface” on page 7-21

sqlfind

Package: database.postgre

Find information about all table types in PostgreSQL database

Syntax

```
data = sqlfind(conn,pattern)
data = sqlfind(conn,pattern,Name,Value)
```

Description

`data = sqlfind(conn,pattern)` returns information about all the "Table Types" on page 11-658 in a database where the specified character pattern appears in the name of the table type. Executing this function is the equivalent of writing the SQL statement `SELECT * FROM information_schema.tables`.

`data = sqlfind(conn,pattern,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, 'Catalog', "cat" finds all table types in the "cat" catalog.

Examples

Find Information About Table Types Using PostgreSQL Native Interface

Use a PostgreSQL native interface database connection to find information about all database table types in a PostgreSQL database.

Create a PostgreSQL native interface database connection to a PostgreSQL database.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";

conn = postgresql(datasource,username,password);
```

Find information about all table types in the database.

```
data = sqlfind(conn,"");
```

Display information about the first three table types.

```
head(data,3)
```

```
ans=3x5 table
      Catalog      Schema      Table      Columns      Type
-----
"toystore_doc"  "information_schema"  "_pg_foreign_data_wrappers"  {1x7 string}  "VI
"toystore_doc"  "information_schema"  "_pg_foreign_servers"       {1x9 string}  "VI
```

```
"toystore_doc"      "information_schema"  "_pg_foreign_table_columns"  {1×4 string}  "VI
```

`data` contains these variables:

- Catalog name
- Schema name
- Table name
- Columns in the table type
- Table type

Close the database connection.

```
close(conn)
```

Find Information About Table Types in Catalog

Use a PostgreSQL native interface database connection to find information about all database table types in a PostgreSQL database. Specify the database catalog to search.

Create a PostgreSQL native interface database connection to a PostgreSQL database.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Find information about all table types in the `toystore_doc` database catalog. Use the 'Catalog' name-value pair argument to specify the catalog. `data` is a table that contains information about all the table types in the specified catalog.

```
data = sqlfind(conn,"",'Catalog','toystore_doc');
```

Display the first eight table types.

```
head(data)
```

```
ans=8×5 table
      Catalog      Schema      Table      Columns
-----
"toystore_doc"  "information_schema"  "_pg_foreign_data_wrappers"  {1×7 string}
"toystore_doc"  "information_schema"  "_pg_foreign_servers"        {1×9 string}
"toystore_doc"  "information_schema"  "_pg_foreign_table_columns"  {1×4 string}
"toystore_doc"  "information_schema"  "_pg_foreign_tables"        {1×7 string}
"toystore_doc"  "information_schema"  "_pg_user_mappings"         {1×7 string}
"toystore_doc"  "information_schema"  "administrable_role_authorizations" {1×3 string}
"toystore_doc"  "information_schema"  "applicable_roles"          {1×3 string}
"toystore_doc"  "information_schema"  "attributes"                 {1×31 string}
```

`data` contains these variables:

- Catalog name
- Schema name
- Table name
- Columns in the database table
- Table type

Display the column names in the fourth table type.

```
data.Columns{4}
```

```
ans = 1x7 string
      "foreign_table_catalog"    "foreign_table_schema"    "foreign_table_name"    "ftoptions"
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

pattern — Pattern

character vector | string scalar

Pattern, specified as a character vector or string scalar. The `sqlfind` function searches for this text in the names of the table types in a database. To find all table types, specify an empty character vector or string scalar.

Example: "inventory"

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `data = sqlfind(conn,pattern,'Catalog','toystore_doc','Schema','dbo')` returns information about table types, stored in the specified catalog and schema, that match the name of the table type with the specified pattern.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: `string | char`

Schema — Database schema name

`string scalar | character vector`

Database schema name, specified as a string scalar or character vector. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Example: `Schema = "dbo"`

Data Types: `string | char`

Output Arguments

data — Table type information

`table`

Table type information, returned as a table that contains information about the table types in the database, where the table type name partially or fully matches the text in `pattern`. The data output contains these variables in a cell array of character vectors.

Variable	Description
Catalog	Catalog name where the database table type is stored
Schema	Schema name where the database table type is stored
Table	Database table name
Columns	Column names in the database table type
Type	Database table type

More About

Table Types

Table types are a subset of database objects that store or reference data.

The `sqlfind` function recognizes these table types in a database:

- Base table
- View
- Foreign table
- Local temporary

Version History

Introduced in R2020b

See Also

`postgresql` | `close` | `sqlread` | `sqlinnerjoin`

Topics

“Import Data from PostgreSQL Database Table” on page 7-11

“Customize Options for Importing Data from PostgreSQL Database into MATLAB” on page 7-13

“Insert Data into Database Table Using PostgreSQL Native Interface” on page 7-19

sqlinnerjoin

Package: database.postgre

Inner join between two PostgreSQL database tables

Syntax

```
data = sqlinnerjoin(conn, lefttable, righttable)
data = sqlinnerjoin(conn, lefttable, righttable, Name, Value)
```

Description

`data = sqlinnerjoin(conn, lefttable, righttable)` returns a table resulting from an inner join between the left and right database tables. This function matches rows using all shared columns, or keys, in both database tables. The inner join retains only the rows that match between the two tables. Executing this function is the equivalent of writing the SQL statement `SELECT * FROM lefttable, righttable INNER JOIN lefttable.key = righttable.key`.

`data = sqlinnerjoin(conn, lefttable, righttable, Name, Value)` uses additional options specified by one or more name-value arguments. For example, specify `Keys = "productNumber"` to use the `productNumber` column as a key for joining the two database tables.

Examples

Join Two Database Tables Using PostgreSQL Native Interface

Use a PostgreSQL native interface database connection to import product data from an inner join between two PostgreSQL database tables into MATLAB®.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource, username, password);
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlinnerjoin` function automatically detects the shared column between the tables. `data` is a table that contains the matched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlinnerjoin(conn, lefttable, righttable);
```

Display the first three rows of matched data. The columns from the right table (`suppliers`) appear to the right of the columns from the left table (`productTable`).


```
head(data,3)
```

```
ans=3x10 table
  productnumber  stocknumber  suppliernumber  unitcost  productdescription  supplier
  _____  _____  _____  _____  _____  _____
           1      4.0035e+05      1001          14      "Building Blocks"      100
           2      4.0031e+05      1002           9      "Painting Set"        100
           3      4.01e+05       1009          17      "Slinky"              100
```

Close the database connection.

```
close(conn)
```

Specify Key for Joining Two Database Tables

Use a PostgreSQL native interface database connection to import joined product data from two PostgreSQL database tables into MATLAB®. Specify the key to use for joining the tables.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. Specify the key, or shared column, between the tables using the 'Keys' name-value pair argument. `data` is a table that contains the matched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlinnerjoin(conn,lefttable,righttable,'Keys',"supplierNumber");
```

Display the first three rows of matched data. The columns from the right table (`suppliers`) appear to the right of the columns from the left table (`productTable`).

```
head(data,3)
```

```
ans=3x10 table
  productnumber  stocknumber  suppliernumber  unitcost  productdescription  supplier
  _____  _____  _____  _____  _____  _____
           1      4.0035e+05      1001          14      "Building Blocks"      100
           2      4.0031e+05      1002           9      "Painting Set"        100
           3      4.01e+05       1009          17      "Slinky"              100
```

Close the database connection.

```
close(conn)
```

Filter Rows in Joined Data

Use a PostgreSQL native interface database connection to import joined product data from two PostgreSQL database tables into MATLAB®. Specify the row filter condition to use for joining the tables.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, username, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Join the two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The table data contains the matched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlinnerjoin(conn,lefttable,righttable);
```

Display the first five rows of matched data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,5)
```

productnumber	stocknumber	suppliernumber	unitcost	productdescription	suppliernumber
1	4.0034e+05	1001	14	"Building Blocks"	1001
2	4.0031e+05	1002	9	"Painting Set"	1002
3	4.01e+05	1009	17	"Slinky"	1009
4	4.0034e+05	1008	21	"Space Cruiser"	1008
5	4.0046e+05	1005	3	"Tin Soldier"	1005

Join the same tables, but this time use a row filter. The filter condition is that `unitCost` must be less than 15. Again, display the first five rows of matched data.

```
rf = rowfilter("unitCost");
rf = rf.unitCost <= 15;
data = sqlinnerjoin(conn,lefttable,righttable,"RowFilter",rf);
head(data,5)
```

productnumber	stocknumber	suppliernumber	unitcost	productdescription	suppliernumber
1	4.0034e+05	1001	14	"Building Blocks"	1001
2	4.0031e+05	1002	9	"Painting Set"	1002
5	4.0046e+05	1005	3	"Tin Soldier"	1005
6	4.0088e+05	1004	8	"Sail Boat"	1004
8	2.1257e+05	1001	5	"Train Set"	1001

Close the database connection.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

lefttable — Left table

character vector | string scalar

Left table, specified as a character vector or string scalar. Specify the name of the database table on the left side of the join.

Example: 'inventoryTable'

Data Types: char | string

righttable — Right table

character vector | string scalar

Right table, specified as a character vector or string scalar. Specify the name of the database table on the right side of the join.

Example: 'productTable'

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `data =`

```
sqlinnerjoin(conn,"productTable","suppliers",'LeftCatalog',"toystore_doc",'LeftSchema',"dbo",'RightCatalog',"toy_shop",'RightSchema',"toys",'MaxRows',5)
```

performs an inner join between left and right tables by specifying the catalog and schema for both tables and returns five matched rows.

LeftCatalog — Left catalog

character vector | string scalar

Left catalog, specified as the comma-separated pair consisting of 'LeftCatalog' and a character vector or string scalar. Specify the database catalog name where the left table of the join is stored.

Example: 'LeftCatalog','toy_store'

Data Types: char | string

RightCatalog — Right catalog

character vector | string scalar

Right catalog, specified as the comma-separated pair consisting of `'RightCatalog'` and a character vector or string scalar. Specify the database catalog name where the right table of the join is stored.

Example: `'RightCatalog', 'toy_store'`

Data Types: `char` | `string`

LeftSchema — Left schema

character vector | string scalar

Left schema, specified as the comma-separated pair consisting of `'LeftSchema'` and a character vector or string scalar. Specify the database schema name where the left table of the join is stored.

Example: `'LeftSchema', 'dbo'`

Data Types: `char` | `string`

RightSchema — Right schema

character vector | string scalar

Right schema, specified as the comma-separated pair consisting of `'RightSchema'` and a character vector or string scalar. Specify the database schema name where the right table of the join is stored.

Example: `'RightSchema', 'dbo'`

Data Types: `char` | `string`

Keys — Keys

character vector | string scalar | cell array of character vectors | string array

Keys, specified as the comma-separated pair consisting of `'Keys'` and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. Use this name-value pair argument to identify the shared keys (columns) between the two tables to join.

You cannot use this name-value pair argument with the `'LeftKeys'` and `'RightKeys'` name-value pair arguments.

Example: `'Keys', 'MANAGER_ID'`

Data Types: `char` | `string` | `cell`

LeftKeys — Left keys

character vector | string scalar | cell array of character vectors | string array

Left keys, specified as the comma-separated pair consisting of `'LeftKeys'` and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the left table for the join to the right table.

Use this name-value pair argument with the `'RightKeys'` name-value pair argument. Both arguments must specify the same number of keys. The `sqlinnerjoin` function pairs the values of the keys based on their order.

Example: `'LeftKeys', ["productNumber" "Price"], 'RightKeys', ["productNumber" "Price"]`

Data Types: `char` | `string` | `cell`

RightKeys — Right keys

character vector | string scalar | cell array of character vectors | string array

Right keys, specified as the comma-separated pair consisting of 'RightKeys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the right table for the join to the left table.

Use this name-value pair argument with the 'LeftKeys' name-value pair argument. Both arguments must specify the same number of keys. The `sqlinnerjoin` function pairs the values of the keys based on their order.

Example: 'LeftKeys',["productIdentifier" "Cost"],'RightKeys',["productNumber" "Price"]

Data Types: char | string | cell

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of 'MaxRows' and a positive numeric scalar. By default, the `sqlinnerjoin` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: 'MaxRows',10

Data Types: double

VariableNamingRule — Variable naming rule

"preserve" (default) | "modify"

Variable naming rule, specified as the comma-separated pair consisting of 'VariableNamingRule' and one of these values:

- "preserve" — Preserve most variable names when the `sqlinnerjoin` function imports data. For details, see the Limitations section.
- "modify" — Remove non-ASCII characters from variable names when the `sqlinnerjoin` function imports data.

Example: 'VariableNamingRule',"modify"

Data Types: string

RowFilter — Row filter condition

<unconstrained> (default) | matlab.io.RowFilter object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; sqlinnerjoin(conn,lefttable,righttable,"RowFilter",rf)`

Output Arguments**data — Joined data**

table

Joined data, returned as a table that contains the matched rows from the join of the left and right tables. `data` also contains a variable for each column in the left and right tables.

If the column names are shared between the joined database tables and have the same case, then the `sqlinnerjoin` function adds a unique suffix to the corresponding variable names in `data`.

When you import data, the `sqlinnerjoin` function converts the data type of each column from the PostgreSQL database to the MATLAB data type. This table maps the data type of a database column to the converted MATLAB data type.

PostgreSQL Data Type	MATLAB Data Type
Boolean	logical
Smallint	double
Integer	double
Bigint	double
Decimal	double
Numeric	double
Real	double
Double precision	double
Smallserial	double
Serial	double
Bigserial	double
Money	double
Varchar	string
Char	string
Text	string
Bytea	string
Timestamp	datetime
Timestampz	datetime
Abstime	datetime
Date	datetime
Time	duration
Timez	duration
Interval	calendarDuration
Reltime	calendarDuration
Enum	categorical
Cidr	string
Inet	string
Macaddr	string
Uuid	string
Xml	string

Limitations

The name-value argument `VariableNamingRule` has these limitations if it is set to the value `"modify"`:

- The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the `table` data type.
- The length of each variable name must be less than the number returned by `namelengthmax`.

Version History

Introduced in R2020b

R2023a: Selectively join data based on filter condition

You can use the `RowFilter` when joining data from database tables.

See Also

`postgresql` | `close` | `sqlread` | `sqlouterjoin` | `sqlfind`

Topics

“Import Data from PostgreSQL Database Table” on page 7-11

“Customize Options for Importing Data from PostgreSQL Database into MATLAB” on page 7-13

sqlouterjoin

Package: database.postgre

Outer join between two PostgreSQL database tables

Syntax

```
data = sqlouterjoin(conn, lefttable, righttable)
data = sqlouterjoin(conn, lefttable, righttable, Name, Value)
```

Description

`data = sqlouterjoin(conn, lefttable, righttable)` returns a table resulting from an outer join between the left and right database tables. This function matches rows using all shared columns, or keys, in both database tables. The outer join retains the matched and unmatched rows between the two tables. Executing this function is the equivalent of writing the SQL statement `SELECT * FROM lefttable, righttable OUTER JOIN lefttable.key = righttable.key`.

`data = sqlouterjoin(conn, lefttable, righttable, Name, Value)` uses additional options specified by one or more name-value arguments. For example, specify `Keys = "productNumber"` to use the `productNumber` column as a key for joining the two database tables.

Examples

Join Two Database Tables Using PostgreSQL Native Interface

Use a PostgreSQL native interface database connection to import product data from an outer join between two PostgreSQL database tables into MATLAB®.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource, username, password);
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The `sqlouterjoin` function automatically detects the shared column between the tables. The `sqlouterjoin` function automatically detects the shared column between the tables. `data` is a table that contains the matched and unmatched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlouterjoin(conn, lefttable, righttable);
```

Display the first three rows of joined data. The columns from the right table (`suppliers`) appear to the right of the columns from the left table (`productTable`).


```
head(data,3)
```

```
ans=3x10 table
  productnumber  stocknumber  suppliernumber  unitcost  productdescription  supplier
  _____  _____  _____  _____  _____  _____
           1      4.0035e+05      1001          14      "Building Blocks"      100
           2      4.0031e+05      1002           9      "Painting Set"        100
           3       4.01e+05      1009          17      "Slinky"              100
```

Close the database connection.

```
close(conn)
```

Specify Key for Joining Two Database Tables

Use a PostgreSQL native interface database connection to import joined product data from two PostgreSQL database tables into MATLAB®. Specify the key to use for joining the tables.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Join two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. Specify the key, or shared column, between the tables using the 'Keys' name-value pair argument. `data` is a table that contains the matched and unmatched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlouterjoin(conn,lefttable,righttable,'Keys',"supplierNumber");
```

Display the first three rows of matched data. The columns from the right table (`suppliers`) appear to the right of the columns from the left table (`productTable`).

```
head(data,3)
```

```
ans=3x10 table
  productnumber  stocknumber  suppliernumber  unitcost  productdescription  supplier
  _____  _____  _____  _____  _____  _____
           1      4.0035e+05      1001          14      "Building Blocks"      100
           2      4.0031e+05      1002           9      "Painting Set"        100
           3       4.01e+05      1009          17      "Slinky"              100
```

Close the database connection.

```
close(conn)
```

Filter Rows in Joined Data

Use a PostgreSQL native interface database connection to import joined product data from two PostgreSQL database tables into MATLAB®. Specify the row filter condition to use for joining the tables.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, username, and password. The database contains the tables `productTable` and `suppliers`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Join the two database tables, `productTable` and `suppliers`. The `productTable` table is the left table of the join, and the `suppliers` table is the right table of the join. The table data contains the matched and unmatched rows from the two tables.

```
lefttable = "productTable";
righttable = "suppliers";
data = sqlouterjoin(conn,lefttable,righttable);
```

Display the first five rows of matched data. The columns from the right table appear to the right of the columns from the left table.

```
head(data,5)
```

productnumber	stocknumber	suppliernumber	unitcost	productdescription	suppliernumber
1	4.0034e+05	1001	14	"Building Blocks"	1001
2	4.0031e+05	1002	9	"Painting Set"	1002
3	4.01e+05	1009	17	"Slinky"	1009
4	4.0034e+05	1008	21	"Space Cruiser"	1008
5	4.0046e+05	1005	3	"Tin Soldier"	1005

Join the same tables, but this time use a row filter. The filter condition is that `unitCost` must be less than 15. Again, display the first five rows of matched data.

```
rf = rowfilter("unitCost");
rf = rf.unitCost <= 15;
data = sqlouterjoin(conn,lefttable,righttable,"RowFilter",rf);
head(data,5)
```

productnumber	stocknumber	suppliernumber	unitcost	productdescription	suppliernumber
1	4.0034e+05	1001	14	"Building Blocks"	1001
2	4.0031e+05	1002	9	"Painting Set"	1002
5	4.0046e+05	1005	3	"Tin Soldier"	1005
6	4.0088e+05	1004	8	"Sail Boat"	1004
8	2.1257e+05	1001	5	"Train Set"	1001

Close the database connection.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

lefttable — Left table

character vector | string scalar

Left table, specified as a character vector or string scalar. Specify the name of the database table on the left side of the join.

Example: 'inventoryTable'

Data Types: char | string

righttable — Right table

character vector | string scalar

Right table, specified as a character vector or string scalar. Specify the name of the database table on the right side of the join.

Example: 'productTable'

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `data =`

```
sqlouterjoin(conn,"productTable","suppliers",'Type','left','MaxRows',5)
performs an outer left join between left and right tables and returns five rows of the joined data.
```

LeftCatalog — Left catalog

character vector | string scalar

Left catalog, specified as the comma-separated pair consisting of 'LeftCatalog' and a character vector or string scalar. Specify the database catalog name where the left table of the join is stored.

Example: 'LeftCatalog','toy_store'

Data Types: char | string

RightCatalog — Right catalog

character vector | string scalar

Right catalog, specified as the comma-separated pair consisting of 'RightCatalog' and a character vector or string scalar. Specify the database catalog name where the right table of the join is stored.

Example: 'RightCatalog','toy_store'

Data Types: char | string

LeftSchema — Left schema

character vector | string scalar

Left schema, specified as the comma-separated pair consisting of 'LeftSchema' and a character vector or string scalar. Specify the database schema name where the left table of the join is stored.

Example: 'LeftSchema','dbo'

Data Types: char | string

RightSchema — Right schema

character vector | string scalar

Right schema, specified as the comma-separated pair consisting of 'RightSchema' and a character vector or string scalar. Specify the database schema name where the right table of the join is stored.

Example: 'RightSchema','dbo'

Data Types: char | string

Keys — Keys

character vector | string scalar | cell array of character vectors | string array

Keys, specified as the comma-separated pair consisting of 'Keys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. Use this name-value pair argument to identify the shared keys (columns) between the two tables to join.

You cannot use this name-value pair argument with the 'LeftKeys' and 'RightKeys' name-value pair arguments.

Example: 'Keys','MANAGER_ID'

Data Types: char | string | cell

LeftKeys — Left keys

character vector | string scalar | cell array of character vectors | string array

Left keys, specified as the comma-separated pair consisting of 'LeftKeys' and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the left table for the join to the right table.

Use this name-value pair argument with the 'RightKeys' name-value pair argument. Both arguments must specify the same number of keys. The `sqlouterjoin` function pairs the values of the keys based on their order.

Example: 'LeftKeys',["productNumber" "Price"],'RightKeys',["productNumber" "Price"]

Data Types: char | string | cell

RightKeys — Right keys

character vector | string scalar | cell array of character vectors | string array

Right keys, specified as the comma-separated pair consisting of `'RightKeys'` and a character vector, string scalar, cell array of character vectors, or string array. Specify a character vector or string scalar to indicate one key. For multiple keys, specify a cell array of character vectors or a string array. This name-value pair argument identifies the keys in the right table for the join to the left table.

Use this name-value pair argument with the `'LeftKeys'` name-value pair argument. Both arguments must specify the same number of keys. The `sqlouterjoin` function pairs the values of the keys based on their order.

Example: `'LeftKeys',["productIdentifier" "Cost"],'RightKeys',["productNumber" "Price"]`

Data Types: `char` | `string` | `cell`

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of `'MaxRows'` and a positive numeric scalar. By default, the `sqlouterjoin` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: `'MaxRows',10`

Data Types: `double`

Type — Outer join type

`'full'` (default) | `'left'` | `'right'`

Outer join type, specified as the comma-separated pair consisting of `'Type'` and one of these values:

- `'full'` — A full join retrieves records that have matching values in the selected column of both tables, and unmatched records from both the left and right tables.
- `'left'` — A left join retrieves records that have matching values in the selected column of both tables, and unmatched records from the left table only.
- `'right'` — A right join retrieves records that have matching values in the selected column of both tables, and unmatched records from the right table only.

You can specify this value as a character vector or string scalar.

Example: `'Type','left'`

VariableNamingRule — Variable naming rule

`"preserve"` (default) | `"modify"`

Variable naming rule, specified as the comma-separated pair consisting of `'VariableNamingRule'` and one of these values:

- `"preserve"` — Preserve most variable names when the `sqlouterjoin` function imports data. For details, see the Limitations section.
- `"modify"` — Remove non-ASCII characters from variable names when the `sqlouterjoin` function imports data.

Example: `'VariableNamingRule',"modify"`

Data Types: `string`

RowFilter — Row filter condition

<unconstrained> (default) | matlab.io.RowFilter object

Row filter condition, specified as a matlab.io.RowFilter object.

Example: rf = rowfilter("productnumber"); rf = rf.productnumber <= 5;
sqlouterjoin(conn,lefttable,righttable,"RowFilter",rf)

Output Arguments**data — Joined data**

table

Joined data, returned as a table that contains rows matched by keys in the left and right database tables and the retained unmatched rows. **data** also contains a variable for each column in the left and right tables.

By default, the variable data types are double for columns that have numeric data types in the database table.

If the column names are shared between the joined database tables and have the same case, then the sqlouterjoin function adds a unique suffix to the corresponding variable names in **data**.

The variables in **data** that correspond to columns in the left table contain NULL values when no matched rows exist in the right database table. Similarly, the variables that correspond to columns in the right table contain NULL values when no matched rows exist in the left database table.

When you import data, the sqlouterjoin function converts the data type of each column from the PostgreSQL database to the MATLAB data type. This table maps the data type of a database column to the converted MATLAB data type.

PostgreSQL Data Type	MATLAB Data Type
Boolean	logical
Smallint	double
Integer	double
Bigint	double
Decimal	double
Numeric	double
Real	double
Double precision	double
Smallserial	double
Serial	double
Bigserial	double
Money	double
Varchar	string
Char	string
Text	string

PostgreSQL Data Type	MATLAB Data Type
Bytea	string
Timestamp	datetime
Timestampz	datetime
Abstime	datetime
Date	datetime
Time	duration
Timez	duration
Interval	calendarDuration
Reltime	calendarDuration
Enum	categorical
Cidr	string
Inet	string
Macaddr	string
Uuid	string
Xml	string

Limitations

The name-value argument `VariableNamingRule` has these limitations if it is set to the value "modify":

- The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the table data type.
- The length of each variable name must be less than the number returned by `namelengthmax`.

Version History

Introduced in R2020b

R2023a: Selectively join data based on filter condition

You can use the `RowFilter` when joining data from database tables.

See Also

`postgresql` | `close` | `sqlread` | `sqlfind` | `sqlinnerjoin`

Topics

"Import Data from PostgreSQL Database Table" on page 7-11

"Customize Options for Importing Data from PostgreSQL Database into MATLAB" on page 7-13

sqlread

Package: database.postgre

Import data into MATLAB from PostgreSQL database table

Syntax

```
data = sqlread(conn,tablename)
data = sqlread(conn,tablename,opts)
data = sqlread( ___,Name,Value)
[data,metadata] = sqlread( ___)
```

Description

`data = sqlread(conn,tablename)` returns a table by importing data into MATLAB from a PostgreSQL database table. Executing this function is the equivalent of writing a `SELECT * FROM tablename` SQL statement in ANSI SQL.

`data = sqlread(conn,tablename,opts)` customizes options for importing data from a database table using the `SQLImportOptions` object.

`data = sqlread(___,Name,Value)` specifies additional options using one or more name-value pair arguments and any of the previous input argument combinations. For example, specify `Catalog = "cat"` to import data from a database table stored in the "cat" catalog.

`[data,metadata] = sqlread(___)` also returns the metadata table, which contains metadata information about the imported data.

Examples

Import Data from Database Table Using PostgreSQL Native Interface

Use a PostgreSQL native interface database connection to import product data from a database table into MATLAB® using a PostgreSQL database. Then, perform a simple data analysis.

Create a PostgreSQL native interface database connection to a PostgreSQL database. The database contains the table `productTable`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
```

```
conn = postgresql(datasource,username,password);
```

Import data from the database table `productTable`. The `sqlread` function returns a MATLAB table that contains the product data.

```
tablename = "productTable";
data = sqlread(conn,tablename);
```


Display the first five rows of product data.

```
head(data,5)
```

productnumber	stocknumber	suppliernumber	unitcost	productdescription
9	1.2597e+05	1003	13	"Victorian Doll"
8	2.1257e+05	1001	5	"Train Set"
7	3.8912e+05	1007	16	"Engine Kit"
2	4.0031e+05	1002	9	"Painting Set"
4	4.0034e+05	1008	21	"Space Cruiser"

Now, import the data using a row filter. The filter condition is that `unitcost` must be less than 15.

```
rf = rowfilter("unitcost");
rf = rf.unitcost < 15;
data = sqlread(conn,tablename,"RowFilter",rf);
```

Again, display the first five products.

```
head(data,5)
```

productnumber	stocknumber	suppliernumber	unitcost	productdescription
9	1.2597e+05	1003	13	"Victorian Doll"
8	2.1257e+05	1001	5	"Train Set"
2	4.0031e+05	1002	9	"Painting Set"
1	4.0034e+05	1001	14	"Building Blocks"
5	4.0046e+05	1005	3	"Tin Soldier"

Close the database connection.

```
close(conn)
```

Import Data from Database Table Using Import Options

Customize import options when importing data from a database table using the PostgreSQL native interface. Control the import options by creating an `SQLImportOptions` object. Then, customize import options for different database columns. Import data using the `sqlread` function.

This example uses the `patients.xls` file, which contains the columns `Gender`, `Location`, `SelfAssessedHealthStatus`, and `Smoker`. The example uses a PostgreSQL database version 9.405 database and the `libpq` driver version 10.12.

Create a PostgreSQL native interface database connection to a PostgreSQL database.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";

conn = postgresql(datasource,username,password);
```

Load patient information into the MATLAB® workspace.

```
patients = readtable("patients.xls");
```

Create the patients database table using the patient information.

```
tablename = "patients";
sqlwrite(conn,tablename,patients)
```

Create an SQLImportOptions object using the patients database table and the databaseImportOptions function.

```
opts = databaseImportOptions(conn,tablename)
```

```
opts =
  SQLImportOptions with properties:

    ExcludeDuplicates: false
    VariableNamingRule: 'preserve'

    VariableNames: {'lastname', 'gender', 'age' ... and 7 more}
    VariableTypes: {'string', 'string', 'double' ... and 7 more}
    SelectedVariableNames: {'lastname', 'gender', 'age' ... and 7 more}
    FillValues: { <missing>, <missing>, NaN ... and 7 more }
    RowFilter: <unconstrained>

    VariableOptions: Show all 10 VariableOptions
```

Display the current import options for the variables in the SelectedVariableNames property of the SQLImportOptions object.

```
vars = opts.SelectedVariableNames;
varOpts = getoptions(opts,vars)
```

```
varOpts =
  1x10 SQLVariableImportOptions array with properties:

    Variable Options:

      (1) | (2) | (3) | (4) | (5) | (6) | (7)
      Name: 'lastname' | 'gender' | 'age' | 'location' | 'height' | 'weight' | 'smoker'
      Type: 'string' | 'string' | 'double' | 'string' | 'double' | 'double' | 'logical'
      MissingRule: 'fill' | 'fill' | 'fill' | 'fill' | 'fill' | 'fill' | 'fill'
      FillValue: <missing> | <missing> | NaN | <missing> | NaN | NaN | 0
```

To access sub-properties of each variable, use getoptions

Change the data types for the gender, location, smoker, and selfassessedhealthstatus variables using the setoptions function. Because the gender, location, and selfassessedhealthstatus variables indicate a finite set of repeating values, change their data type to categorical. Because the Smoker variable stores the values 0 and 1, change its data type to double. Then, display the updated import options.

```
opts = setoptions(opts,{'gender','location','selfassessedhealthstatus'}, ...
  'Type','categorical');
opts = setoptions(opts,'smoker','Type','double');

varOpts = getoptions(opts,{'gender','location','smoker', ...
  'selfassessedhealthstatus'})
```

```

varOpts =
    1x4 SQLVariableImportOptions array with properties:

    Variable Options:
        (1) | (2) | (3) | (4)
        Name: 'gender' | 'location' | 'smoker' | 'selfassessedhealthstatus'
        Type: 'categorical' | 'categorical' | 'double' | 'categorical'
        MissingRule: 'fill' | 'fill' | 'fill' | 'fill'
        FillValue: <undefined> | <undefined> | 0 | <undefined>

```

To access sub-properties of each variable, use `getoptions`

Import the `patients` database table using the `sqlread` function, and display the last eight rows of the table.

```

data = sqlread(conn,tablename,opts);
tail(data)

```

lastname	gender	age	location	height	weight	smoker	sy
"Foster"	Female	30	St. Mary's Medical Center	70	124	0	
"Gonzales"	Male	48	County General Hospital	71	174	0	
"Bryant"	Female	48	County General Hospital	66	134	0	
"Alexander"	Male	25	County General Hospital	69	171	1	
"Russell"	Male	44	VA Hospital	69	188	1	
"Griffin"	Male	49	County General Hospital	70	186	0	
"Diaz"	Male	45	County General Hospital	68	172	1	
"Hayes"	Male	48	County General Hospital	66	177	0	

Display a summary of the imported data. The `sqlread` function applies the import options to the variables in the imported data.

```
summary(data)
```

Variables:

```
lastname: 100x1 string
```

```
gender: 100x1 categorical
```

Values:

```

    Female    53
    Male      47

```

```
age: 100x1 double
```

Values:

```

    Min      25
    Median   39
    Max      50

```

```
location: 100x1 categorical
```

Values:

County General Hospital	39
St. Mary s Medical Center	24
VA Hospital	37

height: 100×1 double

Values:

Min	60
Median	67
Max	72

weight: 100×1 double

Values:

Min	111
Median	142.5
Max	202

smoker: 100×1 double

Values:

Min	0
Median	0
Max	1

systolic: 100×1 double

Values:

Min	109
Median	122
Max	138

diastolic: 100×1 double

Values:

Min	68
Median	81.5
Max	99

selfassessedhealthstatus: 100×1 categorical

Values:

Excellent	34
Fair	15
Good	40
Poor	11

Now set the filter condition to import only data for patients older than 40 year and not taller than 68 inches.

```
opts.RowFilter = opts.RowFilter.Age > 40 & opts.RowFilter.Height <= 68
```

```

opts =
  SQLImportOptions with properties:

    ExcludeDuplicates: false
    VariableNamingRule: 'preserve'

    VariableNames: {'lastname', 'gender', 'age' ... and 7 more}
    VariableTypes: {'string', 'categorical', 'double' ... and 7 more}
    SelectedVariableNames: {'lastname', 'gender', 'age' ... and 7 more}
    FillValues: { <missing>, <undefined>, NaN ... and 7 more }
    RowFilter: height <= 68 & age > 40

    VariableOptions: Show all 10 VariableOptions

```

Again, import the `patients` database table using the `sqlread` function, and display a summary of the imported data.

```

data = sqlread(conn,tablename,opts);
summary(data)

```

Variables:

lastname: 24×1 string

gender: 24×1 categorical

Values:

Female	17
Male	7

age: 24×1 double

Values:

Min	41
Median	45.5
Max	50

location: 24×1 categorical

Values:

County General Hospital	13
St. Mary s Medical Center	5
VA Hospital	6

height: 24×1 double

Values:

Min	62
Median	66
Max	68

weight: 24×1 double

Values:

Min	119
Median	137
Max	194

smoker: 24×1 double

Values:

Min	0
Median	0
Max	1

systolic: 24×1 double

Values:

Min	114
Median	121.5
Max	138

diastolic: 24×1 double

Values:

Min	68
Median	81.5
Max	96

selfassessedhealthstatus: 24×1 categorical

Values:

Excellent	7
Fair	3
Good	10
Poor	4

Delete the `patients` database table using the `execute` function.

```
sqlquery = strcat("DROP TABLE ",tablename);  
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Limit Number of Rows in Imported Data

Use a PostgreSQL native interface database connection to import a limited number of rows of product data from a database table into MATLAB®. Then, sort and filter the rows in the imported data, and perform a simple data analysis.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password. The database contains the table `productTable`.

```

datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);

```

Import data from the table productTable. Limit the number of rows by setting the 'MaxRows' name-value pair argument to 10. The data table contains the product data.

```

tablename = "productTable";
data = sqlread(conn,tablename,'MaxRows',10);

```

Display the first few rows of product data.

```
head(data,3)
```

```

ans=3x5 table
   productnumber  stocknumber  suppliernumber  unitcost  productdescription
   _____  _____  _____  _____  _____
           9      1.2597e+05      1003          13      "Victorian Doll"
           8      2.1257e+05      1001           5      "Train Set"
           7      3.8912e+05      1007          16      "Engine Kit"

```

Display the first few product descriptions.

```
data.productdescription(1:3)
```

```

ans = 3x1 string
    "Victorian Doll"
    "Train Set"
    "Engine Kit"

```

Sort the rows in data by the product description column in alphabetical order.

```

column = "productdescription";
data = sortrows(data,column);

```

Display the first few product descriptions after sorting.

```
data.productdescription(1:3)
```

```

ans = 3x1 string
    "Building Blocks"
    "Engine Kit"
    "Painting Set"

```

Close the database connection.

```
close(conn)
```

Retrieve Metadata Information About Imported Data

Retrieve metadata information when importing data from a database table using the PostgreSQL native interface. Import data using the `sqlread` function and explore the metadata information by using dot notation.

This example uses the `outages.csv` file, which contains outage data. The example uses a PostgreSQL database version 9.4.05 database and the libpq driver version 10.12.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Load outage information into the MATLAB® workspace.

```
outages = readtable("outages.csv");
```

Create the `outages` database table using the outage information. Use the 'ColumnType' name-value pair argument to specify the data types of the variables in the MATLAB® table.

```
tablename = "outages";
sqlwrite(conn,tablename,outages, ...
    'ColumnType',["varchar(120)","timestamp","numeric(38,16)", ...
    "numeric(38,16)","timestamp","varchar(150)"])
```

Import the data into the MATLAB workspace and return metadata information about the imported data.

```
[data,metadata] = sqlread(conn,tablename);
```

View the names of the variables in the imported data.

```
metadata.Properties.RowNames
```

```
ans = 6x1 cell
    {'region'      }
    {'outagetime'  }
    {'loss'        }
    {'customers'   }
    {'restorationtime'}
    {'cause'       }
```

View the data type of each variable in the imported data.

```
metadata.VariableType
```

```
ans = 6x1 cell
    {'string' }
    {'datetime'}
    {'double' }
    {'double' }
    {'datetime'}
    {'string' }
```


View the missing data value for each variable in the imported data.

```
metadata.FillValue
```

```
ans=6x1 cell array
    {1x1 missing}
    {[NaT      ]}
    {[      NaN]}
    {[      NaN]}
    {[NaT      ]}
    {1x1 missing}
```

View the indices of the missing data for each variable in the imported data.

```
metadata.MissingRows
```

```
ans=6x1 cell array
    { 0x1 double}
    { 0x1 double}
    {604x1 double}
    {328x1 double}
    { 29x1 double}
    { 0x1 double}
```

Display the first eight rows of the imported data that contain missing restoration time values. `data` contains restoration time values in the fifth variable. Use the numeric indices to find the rows with missing data.

```
index = metadata.MissingRows{5,1};
nullrestoration = data(index,:);
head(nullrestoration)
```

```
ans=8x6 table
    region      outagetime      loss      customers      restorationtime      cause
    _____  _____  _____  _____  _____  _____
    "SouthEast"  23-Jan-2003 00:49:00  530.14  2.1204e+05  NaT  "winter sto
    "NorthEast"  18-Sep-2004 05:54:00      0      0  NaT  "equipment
    "MidWest"    20-Apr-2002 16:46:00  23141  NaN  NaT  "unknown"
    "NorthEast"  16-Sep-2004 19:42:00  4718  NaN  NaT  "unknown"
    "SouthEast"  14-Sep-2005 15:45:00  1839.2  3.4144e+05  NaT  "severe sto
    "SouthEast"  17-Aug-2004 17:34:00  624.1  1.7879e+05  NaT  "severe sto
    "SouthEast"  28-Jan-2006 23:13:00  498.78  NaN  NaT  "energy eme
    "West"       20-Jun-2003 18:22:00      0      0  NaT  "energy eme
```

Delete the outages database table using the `execute` function.

```
sqlstr = "DROP TABLE ";
sqlquery = strcat(sqlstr,tablename);
execute(conn,sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

opts — Database import options

SQLImportOptions object

Database import options, specified as an SQLImportOptions object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `data =`

```
sqlread(conn, "inventoryTable", 'Catalog', "toystore_doc", 'Schema', "dbo", 'MaxRows', 5)
```

imports five rows of data from the database table `inventoryTable` stored in the `toystore_doc` catalog and the `dbo` schema.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: string | char

Schema — Database schema name

string scalar | character vector

Database schema name, specified as a string scalar or character vector. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Example: `Schema = "dbo"`

Data Types: string | char

MaxRows — Maximum number of rows to return

positive numeric scalar

Maximum number of rows to return, specified as the comma-separated pair consisting of 'MaxRows' and a positive numeric scalar. By default, the `sqlread` function returns all rows from the executed SQL query. Use this name-value pair argument to limit the number of rows imported into MATLAB.

Example: 'MaxRows', 10

Data Types: double

VariableNamingRule — Variable naming rule

"preserve" (default) | "modify"

Variable naming rule, specified as the comma-separated pair consisting of 'VariableNamingRule' and one of these values:

- "preserve" — Preserve most variable names when the `sqlread` function imports data. For details, see the Limitations section.
- "modify" — Remove non-ASCII characters from variable names when the `sqlread` function imports data.

Example: 'VariableNamingRule', "modify"

Data Types: string

RowFilter — Row filter condition

<unconstrained> (default) | matlab.io.RowFilter object

Row filter condition, specified as a `matlab.io.RowFilter` object.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5; sqlread(conn, tablename, "RowFilter", rf)`

Output Arguments**data — Imported data**

table

Imported data, returned as a table. The rows of the table correspond to the rows in the database table `tablename`. The variables in the table correspond to each column in the database table.

If the database table contains no data to import, then `data` is an empty table.

When you import data, the `sqlread` function converts the data type of each column from the PostgreSQL database to the MATLAB data type. This table maps the data type of a database column to the converted MATLAB data type.

PostgreSQL Data Type	MATLAB Data Type
Boolean	logical
Smallint	double
Integer	double
Bigint	double

PostgreSQL Data Type	MATLAB Data Type
Decimal	double
Numeric	double
Real	double
Double precision	double
Smallserial	double
Serial	double
Bigserial	double
Money	double
Varchar	string
Char	string
Text	string
Bytea	string
Timestamp	datetime
Timestampz	datetime
Abstime	datetime
Date	datetime
Time	duration
Timez	duration
Interval	calendarDuration
Reltime	calendarDuration
Enum	categorical
Cidr	string
Inet	string
Macaddr	string
Uuid	string
Xml	string

metadata – Metadata information

table

Metadata information, returned as a table with these variables.

Variable Name	Variable Description	Variable Data Type
VariableType	Data type of each variable in the imported data	Cell array of character vectors
FillValue	Value of missing data for each variable in the imported data	Cell array of missing data values
MissingRows	Indices for each occurrence of missing data in each variable of the imported data	Cell array of numeric indices

By default, the `sqlread` function imports text data as a character vector and numeric data as a double. `FillValue` is an empty character array (for text data) or NaN (for numeric data) by default. To change the missing data value to another value, use the `SQLImportOptions` object.

The `RowNames` property of the metadata table contains the names of the variables in the imported data.

Limitations

- The `sqlread` function returns an error when you use the `VariableNamingRule` name-value argument with the `SQLImportOptions` object `opts`.
- When the `VariableNamingRule` name-value pair argument is set to the value "modify":
 - The variable names `Properties`, `RowNames`, and `VariableNames` are reserved identifiers for the table data type.
 - The length of each variable name must be less than the number returned by `namelengthmax`.
- The `sqlread` function returns an error if you specify the `RowFilter` name-value argument with the `SQLImportOptions` object `opts`. It is ambiguous which of the `RowFilter` object to use in this case, especially if the filter conditions are different.

Version History

Introduced in R2020b

R2023a: Selectively import rows of data based on filter condition

You can use the `RowFilter` name-value argument to selectively import rows of data from a database table.

See Also

Functions

`postgresql` | `close` | `sqlfind` | `sqlinnerjoin` | `sqlouterjoin` | `fetch` | `databaseImportOptions` | `getoptions` | `setoptions` | `reset` | `execute`

Topics

"Import Data from PostgreSQL Database Table" on page 7-11

"Customize Options for Importing Data from PostgreSQL Database into MATLAB" on page 7-13

sqlupdate

Package: database.postgre

Update rows in PostgreSQL database table

Syntax

```
sqlupdate(conn,tablename,data,filter)
sqlupdate( ____,Name,Value)
```

Description

`sqlupdate(conn,tablename,data,filter)` updates rows in the PostgreSQL database table (`tablename`) with the rows from the MATLAB table (`data`) based on filter conditions (`filter`).

`sqlupdate(____,Name,Value)` specifies additional options using one or more name-value arguments with any of the previous input argument combinations. For example, `Catalog = "cat"` updates data from a database table stored in the "cat" catalog.

Examples

Update Database Rows

Update rows in the PostgreSQL native interface database based on filter conditions specified with row filters.

This example uses the `patients.xls` file, which contains the columns `LastName`, `Gender`, `Age`, `Location`, `Height`, `Weight`, `Smoker`, `Systolic`, `Diastolic`, and `SelfAssessedHealthStatus`. The example uses a PostgreSQL database version 9.4.05 database and the `libpq` driver version 10.12.

Create a PostgreSQL native interface database connection to a PostgreSQL database.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";

conn = postgresql(datasource,username,password);
```

Load patient information into the MATLAB workspace.

```
patients = readtable("patients.xls");
```

Create the `patients` database table using the patient information.

```
tablename = "patients";
sqlwrite(conn,tablename,patients)
```

Use the SQL `ALTER` statement to add the column `HighRisk` to the table `patients`.

```
sqlquery = 'ALTER TABLE patients ADD HighRisk boolean';
execute(conn,sqlquery)
```

Import the `patients` database table using the `sqlread` function, and return metadata information about the imported data.

```
[data,metadata] = sqlread(conn,tablename);
```

Display the first 10 rows of the table. In MATLAB, all the values in the `HighRisk` column appear as `false`.

```
head(data,10)
```

lastname	gender	age	location	height	weight	smoker
"Smith"	"Male"	38	"County General Hospital"	71	176	true
"Johnson"	"Male"	43	"VA Hospital"	69	163	false
"Williams"	"Female"	38	"St. Mary's Medical Center"	64	131	false
"Jones"	"Female"	40	"VA Hospital"	67	133	false
"Brown"	"Female"	49	"County General Hospital"	64	119	false
"Davis"	"Female"	46	"St. Mary's Medical Center"	68	142	false
"Miller"	"Female"	33	"VA Hospital"	64	142	true
"Wilson"	"Male"	40	"VA Hospital"	68	180	false
"Moore"	"Male"	28	"St. Mary's Medical Center"	68	183	false
"Taylor"	"Female"	31	"County General Hospital"	66	132	false

Displaying the metadata shows that the values are `NULL` (missing elements) in the database.

```
metadata
```

```
metadata=11x3 table
```

	VariableType	FillValue	MissingRows
lastname	{'string' }	{1x1 missing}	{ 0x1 double}
gender	{'string' }	{1x1 missing}	{ 0x1 double}
age	{'double' }	{[NaN]}	{ 0x1 double}
location	{'string' }	{1x1 missing}	{ 0x1 double}
height	{'double' }	{[NaN]}	{ 0x1 double}
weight	{'double' }	{[NaN]}	{ 0x1 double}
smoker	{'logical' }	{[0]}	{ 0x1 double}
systolic	{'double' }	{[NaN]}	{ 0x1 double}
diastolic	{'double' }	{[NaN]}	{ 0x1 double}
selfassessedhealthstatus	{'string' }	{1x1 missing}	{ 0x1 double}
highrisk	{'logical' }	{[0]}	{100x1 double}

Now, identify patients who are considered high risk for developing some hypothetical health issue based on their age and their smoker status. First, create a table containing the new data to write to the database. This table requires only `true` and `false` values.

```
t = table([true;false], "VariableNames", "HighRisk");
head(t)
```

```
HighRisk
-----
true
false
```

Create a row filter using the filter condition that a patient must be older than 35 years and a smoker to be considered high-risk.

```
rf = rowfilter(["Age", "Smoker"]);
rf = rf.Age > 35 & rf.Smoker == true
```

```
rf =
  RowFilter with constraints:
    Age > 35 & Smoker == true
  VariableNames: Age, Smoker
```

Update the `HighRisk` column using this filter to set the values to `true` and using the `~rf` value of the filter to set the value to `false`.

```
sqlupdate(conn, "patients", t, {rf; ~rf});
```

Again, import the `patients` database table using the `sqlread` function, and display the first 10 rows.

```
data = sqlread(conn, tablename);
head(data, 10)
```

lastname	gender	age	location	height	weight	smoker
"Smith"	"Male"	38	"County General Hospital"	71	176	true
"White"	"Male"	39	"VA Hospital"	72	202	true
"Martin"	"Male"	48	"VA Hospital"	71	181	true
"Lee"	"Female"	44	"County General Hospital"	66	146	true
"Wright"	"Female"	45	"VA Hospital"	70	126	true
"Baker"	"Male"	44	"VA Hospital"	71	192	true
"Mitchell"	"Male"	39	"County General Hospital"	71	164	true
"Roberts"	"Male"	44	"VA Hospital"	70	169	true
"Turner"	"Male"	37	"VA Hospital"	70	194	true
"Collins"	"Male"	42	"County General Hospital"	67	179	true

Delete the `patients` database table using the `execute` function.

```
sqlquery = strcat("DROP TABLE ", tablename);
execute(conn, sqlquery)
```

Close the database connection.

```
close(conn)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as an ODBC connection object or JDBC connection object created using the `database` function.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

data — Updated data

MATLAB table

Updated data, specified as a MATLAB table. The table can contain one or more rows with updated data. The names of the variables in the table must be a subset of the column names of the database table.

Example: `data = table([1;0], "VariableNames", "NewName")`

Data Types: table

filter — Row filter condition

matlab.io.RowFilter object | cell array of matlab.io.RowFilter objects

Row filter condition, specified as a `matlab.io.RowFilter` object or cell array of `matlab.io.RowFilter` objects. Filters determine which database rows `sqlupdate` must update with which data. If multiple database rows match a filter, `sqlupdate` updates them with the same data. If a single database row matches multiple filters, its final state matches the data corresponding to the last matching filter.

Example: `rf = rowfilter("productnumber"); rf = rf.productnumber <= 5;`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `sqlupdate(conn, 'inventoryTable', data, rf, Catalog = "toy_store", Schema = "dbo")` updates the database `inventoryTable` stored in the `toy_store` catalog and the `dbo` schema.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: string | char

Schema — Database schema name

string scalar | character vector

Database schema name, specified as a string scalar or character vector. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Example: `Schema = "dbo"`

Data Types: `string` | `char`

Version History

Introduced in R2023a

See Also

`postgresql` | `close` | `sqlread` | `sqlfind` | `sqlinnerjoin` | `sqlouterjoin` | `fetch` | `databaseImportOptions` | `getoptions` | `setoptions` | `reset` | `execute`

sqlwrite

Package: database.postgre

Insert MATLAB data into PostgreSQL database table

Syntax

```
sqlwrite(conn,tablename,data)
sqlwrite(conn,tablename,data,Name,Value)
```

Description

`sqlwrite(conn,tablename,data)` inserts data from a MATLAB table into a database table. If the table exists in the database, this function appends the data from the MATLAB table as rows in the existing database table. If the table does not exist in the database, this function creates a table with the specified table name and then inserts the data as rows in the new table. This syntax is the equivalent of executing SQL statements that contain the CREATE TABLE and INSERT INTO ANSI SQL syntaxes.

`sqlwrite(conn,tablename,data,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, 'Catalog','toystore_doc' inserts data into a database table that is located in the database catalog named toystore_doc.

Examples

Append Data into Existing Table Using PostgreSQL Native Interface

Use a PostgreSQL native interface database connection to append product data from a MATLAB® table into an existing table in a PostgreSQL database.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password. The database contains the table `productTable`.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

To view the existing database table `productTable` before appending data, import its contents into MATLAB and display the last few rows.

```
tablename = "productTable";
rows = sqlread(conn,tablename);
tail(rows,3)
```

```
ans=3x5 table
   productnumber   stocknumber   suppliernumber   unitcost   productdescription
   _____   _____   _____   _____   _____
           6       4.0088e+05       1004           8       "Sail Boat"
```

3	4.01e+05	1009	17	"Slinky"
10	8.8865e+05	1006	24	"Teddy Bear"

Create a MATLAB table that contains the data for one product.

```
data = table(30,500000,1000,25,"Rubik's Cube", ...
    'VariableNames',["productnumber" "stocknumber" ...
    "suppliernumber" "unitcost" "productdescription"]);
```

Append the product data into the database table productTable.

```
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB again and display the last few rows. The results contain a new row for the inserted product.

```
rows = sqlread(conn,tablename);
tail(rows,4)
```

```
ans=4x5 table
    productnumber    stocknumber    suppliernumber    unitcost    productdescription
    _____    _____    _____    _____    _____
         6         4.0088e+05         1004             8         "Sail Boat"
         3         4.01e+05         1009             17         "Slinky"
        10         8.8865e+05         1006             24         "Teddy Bear"
        30             5e+05         1000             25         "Rubik's Cube"
```

Close the database connection.

```
close(conn)
```

Insert Data into New Table

Use a PostgreSQL native interface database connection to insert product data from MATLAB® into a new table in a PostgreSQL database.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Create a MATLAB table that contains data for two products.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
    ["Rubik's Cube";"Doll House"],'VariableNames',["productnumber" ...
    "stocknumber" "suppliernumber" "unitcost" "productdescription"]);
```

Insert the product data into a new database table named toytable.

```
tablename = "toytable";
sqlwrite(conn,tablename,data)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
  productnumber  stocknumber  suppliernumber  unitcost  productdescription
  _____  _____  _____  _____  _____
           30           5e+05           1000           25           "Rubik's Cube"
           40           6e+05           2000           30           "Doll House"
```

Close the database connection.

```
close(conn)
```

Specify Column Types When Inserting Data into New Table

Use a PostgreSQL native interface database connection to insert product data from MATLAB® into a new table in a PostgreSQL database. Specify the data types of the columns in the new database table.

Create a PostgreSQL native interface database connection to a PostgreSQL database using the data source name, user name, and password.

```
datasource = "PostgreSQLDataSource";
username = "dbdev";
password = "matlab";
conn = postgresql(datasource,username,password);
```

Create a MATLAB table that contains data for two products.

```
data = table([30;40],[500000;600000],[1000;2000],[25;30], ...
  ["Rubik's Cube";"Doll House"],'VariableNames',["productnumber" ...
  "stocknumber" "suppliernumber" "unitcost" "productdescription"]);
```

Insert the product data into a new database table named toytable. Use the 'ColumnType' name-value pair argument and a string array to specify the data types of all the columns in the database table.

```
tablename = "toytable";
coltypes = ["numeric" "numeric" "numeric" "numeric" "varchar(255)"];
sqlwrite(conn,tablename,data,'ColumnType',coltypes)
```

Import the contents of the database table into MATLAB and display the rows. The results contain two rows for the inserted products.

```
rows = sqlread(conn,tablename)
```

```
rows=2x5 table
  productnumber  stocknumber  suppliernumber  unitcost  productdescription
  _____  _____  _____  _____  _____
```

30	5e+05	1000	25	"Rubik's Cube"
40	6e+05	2000	30	"Doll House"

Close the database connection.

```
close(conn)
```

Input Arguments

conn — PostgreSQL native interface database connection

connection object

PostgreSQL native interface database connection, specified as a connection object.

tablename — Database table name

string scalar | character vector

Database table name, specified as a string scalar or character vector denoting the name of a table in the database.

Example: "employees"

Data Types: string | char

data — Data to insert

table

Data to insert into a database table, specified as a table.

The valid data types in a MATLAB table are:

- Numeric array
- Cell array of numeric arrays
- Cell array of character vectors
- String array
- Datetime array
- Duration array
- Calendar duration array
- Logical array
- Cell array of logical arrays

The numeric array can contain these data types:

- int8
- uint8
- int16
- uint16
- int32
- uint32

- int64
- uint64
- single
- double

For date and time data, supported formats are:

- Date — 'yyyy-MM-dd'
- Time — 'hh:mm:ss'
- Timestamp — 'yyyy-MM-dd HH:mm:ss'

If the date and time data is specified in an invalid format, then the `sqlwrite` function automatically converts the data to a supported format.

If the cell array of character vectors or string array is specified in an invalid format, then the `sqlwrite` function enables the database driver to check the format. If the format is unexpected, then the database driver throws an error.

You can insert data in an existing database table or a new database table. The data types of variables in `data` vary depending on whether the database table exists. For valid data types, see “Data Types for Existing Table” on page 11-0 and “Data Types for New Table” on page 11-0 .

Note The `sqlwrite` function supports only the `table` data type for the `data` input argument. To insert data stored in a structure, cell array, or numeric matrix, convert the data to a `table` by using the `struct2table`, `cell2table`, and `array2table` functions, respectively.

To insert missing data, see “Accepted Missing Data” on page 11-0 .

Example: `table([10;20],{'M';'F'})`

Data Types for Existing Table

The variable names of the MATLAB table must match the column names in the database table. The `sqlwrite` function is case-sensitive.

When you insert data into a database table, use the data types shown in the following table to ensure that the data has the correct data type. This table matches the valid data types of the MATLAB table variable to the data types of the database column. For example, when you insert data into a database column that has the BIT data type, ensure that the corresponding variable in the MATLAB table is a logical array or cell array of logical arrays.

Data Type of MATLAB Table Variable	Data Type of Existing Database Column
Numeric array or cell array of numeric arrays	<ul style="list-style-type: none"> • smallint • integer • bigint • decimal • numeric • real • double precision • smallserial • serial • bigserial
Cell array of character vectors, string array, datetime array, or duration array	<ul style="list-style-type: none"> • date • time • timestamp
Calendar duration array	interval
Logical array or cell array of logical arrays	bit
Cell array of character vectors or string array	<ul style="list-style-type: none"> • char • varchar

Data Types for New Table

The specified table name for the new database table must be unique across all tables in the database.

The valid data types in a MATLAB table are:

- Numeric array
- Cell array of character vectors
- String array
- Datetime array
- Duration array
- Calendar duration array
- Logical array

The `sqlwrite` function ignores any invalid data types and inserts only the valid variables from MATLAB as columns in a new database table.

The `sqlwrite` function converts the data type of the variable into the default data type of the column in the database table. The following table matches the valid data types of the MATLAB table variable to the default data types of the database column.

Data Type of MATLAB Table Variable	Default Data Type of Database Column
int8 array	smallint
int16 array	smallint

Data Type of MATLAB Table Variable	Default Data Type of Database Column
int32 array	integer
int64 array	bigint
logical array	boolean
single or double array	numeric
datetime array	timestamp
duration array	time
calendarDuration array	interval
cell array of character vectors or string array	varchar
	Note The size of this column equals the sum of the maximum length of a string in the string array and 100.

To specify database-specific column data types instead of the defaults, use the 'ColumnType' name-value pair argument. For example, you can specify 'ColumnType', "bigint" to create a **bigint** column in the new database table.

Also, using the 'ColumnType' name-value pair argument, you can specify other data types that are not in the default list. For example, to insert currency, specify 'ColumnType', "money".

Accepted Missing Data

The accepted missing data for inserting data into a database depends on the data type of the MATLAB table variable and the data type of the column in the database. The following table matches the data type of the MATLAB table variable to the data type of the database column and specifies the accepted missing data to use in each case.

Data Type of MATLAB Table Variable	Data Type of Database Column	Accepted Missing Data
datetime array	date or timestamp	NaT
duration array	time	NaN
calendarDuration array	interval	NaN
double or single array or cell array of double or single arrays	numeric	NaN, [], or ''
cell array of character vectors	date or timestamp	'NaT' or ''
cell array of character vectors	time	'NaN' or ''
cell array of character vectors	char, varchar, or other text data type	''
string array	date or timestamp	"", "NaT", or missing
string array	time	"", "NaN", or missing
string array	char, varchar, or other text data type	missing

Data Types: table

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `sqlwrite(conn, 'tablename', data, 'ColumnType', ["numeric" "timestamp" "money"])` inserts data into a new database table named `tablename` by specifying data types for all columns in the new database table.

Catalog — Database catalog name

string scalar | character vector

Database catalog name, specified as a string scalar or character vector. A catalog serves as the container for the schemas in a database and contains related metadata information. A database can have multiple catalogs.

Example: `Catalog = "toy_store"`

Data Types: string | char

Schema — Database schema name

string scalar | character vector

Database schema name, specified as a string scalar or character vector. A schema defines the database tables, views, relationships among tables, and other elements. A database catalog can have numerous schemas.

Example: `Schema = "dbo"`

Data Types: string | char

ColumnType — Database column types

character vector | string scalar | cell array of character vectors | string array

Database column types, specified as the comma-separated pair consisting of `'ColumnType'` and a character vector, string scalar, cell array of character vectors, or string array. Use this name-value pair argument to define custom data types for the columns in a database table. Specify a column type for each column in the table.

Example: `'ColumnType', ["numeric" "varchar(400)"]`

Data Types: char | string | cell

Version History

Introduced in R2020b

See Also

`sqlread` | `postgresql` | `close` | `cell2table` | `array2table` | `struct2table`

Topics

“Insert Data into Database Table Using PostgreSQL Native Interface” on page 7-19

Neo4jConnect

Neo4j database connection

Description

Create a Neo4j database connection using the MATLAB interface to Neo4j with the REST API. With the Neo4j database connection, you can explore the graph database, update the graph database, store a MATLAB directed graph, and perform graph analytics using the MATLAB directed graph.

You can also create a Neo4j database connection using the Database Toolbox Interface for Neo4j Bolt Protocol. To use this interface, you must install the Database Toolbox Interface for Neo4j Bolt Protocol. For details, see “Database Toolbox Interface for Neo4j Bolt Protocol Installation” on page 9-37.

With a Neo4jConnect object, you can perform these tasks:

- Explore the graph database for nodes and relationships.
- Search the graph database for nodes, relationships, or a subgraph.
- Store a directed graph.
- Update nodes and relationships in the graph database.
- Execute a Cypher query.

Creation

Create a Neo4jConnect object using `neo4j`.

Properties

URL — Neo4j database connection URL

character vector

This property is read-only.

Neo4j database connection URL that contains the server, port number, and web location of the Neo4j database, specified as a character vector.

If you specify a Bolt database connection URL using the Database Toolbox Interface for Neo4j Bolt Protocol, then the `neo4j` function creates a Bolt connection instead.

Example: `http://localhost:7474/db/data` specifies using the HTTP protocol where `localhost` is the server, `7474` is the port number, and `/db/data` is the web location of the database.

Example: `bolt://localhost:7687/db/data` specifies using the Bolt protocol where `localhost` is the server, `7687` is the port number, and `/db/data` is the web location of the database.

Data Types: `char`

UserName — User name

character vector

This property is read-only.

User name for accessing the Neo4j database, specified as a character vector.

Data Types: char

Message — Error message

character vector

This property is read-only.

Error message, specified as a character vector. If this property is empty, the database connection is successful.

Data Types: char

Object Functions**Graph Database Connection**

close Close Neo4j database connection

Traverse the Graph

nodeLabels	All node labels in Neo4j database
relationTypes	All relationship types in Neo4j database
propertyKeys	All property keys in Neo4j database
searchNodeByID	Search Neo4j database nodes by node identifier
searchNode	Search Neo4j database nodes by label or by property key and value
searchRelation	Search relationships for Neo4j database node
searchRelationByID	Search Neo4j relationship by relationship identifier
searchGraph	Search for subgraph or entire graph in Neo4j database

Update the Graph

createNode	Create nodes in Neo4j database
createRelation	Create relationships between nodes in Neo4j database
deleteNode	Delete nodes from Neo4j database
deleteRelation	Delete relationships from Neo4j database

Update Labels and Properties in Graph

addNodeLabel	Add labels to nodes in Neo4j database
removeNodeLabel	Remove labels from nodes in Neo4j database
removeNodeProperty	Remove properties from nodes in Neo4j database
removeRelationProperty	Remove properties from relationships in Neo4j database
setNodeProperty	Set properties for nodes in Neo4j database
setRelationProperty	Set properties for relationships in Neo4j database
updateNode	Update node labels and properties in Neo4j database
updateRelation	Update relationship properties in Neo4j database

Store Directed Graph

storeDigraph Store directed graph in Neo4j database

Execute Cypher Query

executeCypher Execute Cypher query on Neo4j database

Examples

Connect to Neo4j Database

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password)
```

```
neo4jconn =
  Neo4jConnect with properties:
      URL: 'http://localhost:7474/db/data/'
  UserName: 'neo4j'
  Message: []
```

`neo4j` returns a `Neo4jConnect` object with these properties:

- `URL` — The Neo4j database web location
- `UserName` — The user name used to connect to the database
- `Message` — Any database connection error messages

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful Neo4j database connection.

```
neo4jconn.Message
```

```
ans =
     []
```

Retrieve all node labels using the Neo4j database connection `neo4jconn`. The cell array `nlabels` contains a character vector for the one node label in the Neo4j database.

```
nlabels = nodeLabels(neo4jconn)

nlabels = 1x1 cell array
    {'Person'}
```

Close the database connection.

```
close(neo4jconn)
```

Version History

Introduced in R2016b

See Also

Topics

“Determine Dependencies of Services in Network” on page 9-22

“Find Shortest Path Between People in Social Neighborhood” on page 9-27

“Find Friends of Friends in Social Neighborhood” on page 9-32

“Explore Graph Database Structure” on page 9-2

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Search Graph Database” on page 9-9

Neo4jNode

Neo4j database node

Description

After creating a Neo4j database connection using the MATLAB interface to Neo4j, explore nodes in the database. With a `Neo4jNode` object, you can explore the node degree and relationship types of the nodes in the database.

Creation

Create a `Neo4jNode` object using the `createNode`, `searchNodeByID`, and `searchNode` functions.

Properties

NodeID — Node identifier

double

This property is read-only.

Node identifier for the unique node in the Neo4j database, specified as a double.

Data Types: double

NodeData — Node data

structure

This property is read-only.

Node data consisting of property keys and values for the unique node in the Neo4j database, specified as a structure.

Data Types: struct

NodeLabels — Node labels

character vector | cell array of character vectors

This property is read-only.

Node labels of the unique Neo4j database node, specified as a character vector for one label or as a cell array of character vectors for multiple labels.

Data Types: char | cell

Object Functions

<code>nodeDegree</code>	In-degree and out-degree for each associated relationship type for Neo4j database node
<code>nodeRelationTypes</code>	Associated relationship types for Neo4j database node

Examples

Search Neo4j® Database by Node Identifier

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';  
username = 'neo4j';  
password = 'matlab';
```

```
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j® connection object `neo4jconn`.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

The blank `Message` property indicates a successful connection.

Search the database for the node with the node identifier 2 by using the Neo4j database connection `neo4jconn`.

```
nodeid = 2;
```

```
nodeinfo = searchNodeByID(neo4jconn,nodeid)
```

```
nodeinfo =
```

```
    Neo4jNode with properties:
```

```
        NodeID: 2  
        NodeData: [1x1 struct]  
        NodeLabels: 'Person'
```

`nodeinfo` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

Access the property keys and values of the node using the property `NodeData`.

```
nodeinfo.NodeData
```

```
ans =
```

```
    struct with fields:
```


name: 'User2'

Version History

Introduced in R2016b

See Also

Topics

“Explore Graph Database Structure” on page 9-2

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Search Graph Database” on page 9-9

Neo4jRelation

Neo4j database relationship

Description

The `Neo4jRelation` object represents a single relationship in a Neo4j database. Use this object to find information about the relationship between two nodes in a Neo4j database.

Creation

Create a `Neo4jRelation` object using the `createRelation`, `searchRelation`, and `searchRelationByID` functions.

Properties

RelationID — Relationship identifier

numeric scalar

This property is read-only.

Relationship identifier, specified as a numeric scalar. The Neo4j database assigns this number automatically.

Example: 3

Data Types: `double`

RelationData — Relationship data

structure

This property is read-only.

Relationship data consisting of property keys and values for the unique relationship in the Neo4j database, specified as a structure. If the relationship has no properties, then this structure contains no fields.

Data Types: `struct`

StartNodeID — Start node identifier

numeric scalar

This property is read-only.

Start node identifier, specified as a numeric scalar. This number specifies the start node of the Neo4j database relationship.

Example: 3

Data Types: `double`

RelationshipType — Relationship type

character vector

This property is read-only.

Relationship type, specified as a character vector. This character vector specifies the type of the Neo4j database relationship.

Example: 'knows'

Data Types: char

EndNodeID — End node identifier

numeric scalar

This property is read-only.

End node identifier, specified as a numeric scalar. This number specifies the end node of the Neo4j database relationship.

Example: 7

Data Types: double

Examples**Search Neo4j Database by Relationship Identifier**

Search for information about a Neo4jRelation object in a Neo4j® database and display the information.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with values User1 through User7. Each relationship has the type knows.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Search the database for the relationship with the identifier 3 by using the Neo4j database connection.

```
relationid = 3;
relinfo = searchRelationByID(neo4jconn,relationid)

relinfo =

    Neo4jRelation with properties:

        RelationID: 3
        RelationData: [1x1 struct]
        StartNodeID: 1
        RelationType: 'knows'
        EndNodeID: 3
```

`relinfo` is a `Neo4jRelation` object with these properties:

- Relationship identifier
- Relationship data
- Start node identifier
- Relationship type
- End node identifier

Access the property keys and values of the relationship using the property `RelationData`. Here, the relationship does not contain properties, so the structure has no fields.

```
relinfo.RelationData
```

```
ans =

    struct with no fields.
```

Version History

Introduced in R2018a

See Also

Topics

“Explore Graph Database Structure” on page 9-2

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

neo4j

Connect to Neo4j database

Syntax

```
neo4jconn = neo4j(url,username,password)
```

Description

The `neo4j` function creates connections to a Neo4j database. For relational database connections, see “Connect to Database” on page 2-121.

`neo4jconn = neo4j(url,username,password)` creates a `Neo4jConnect` object using the URL, user name, and password for the Neo4j database. Use the object to retrieve graph data from the Neo4j database.

Examples

Connect to Neo4j Database

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password)

neo4jconn =
    Neo4jConnect with properties:
        URL: 'http://localhost:7474/db/data/'
        UserName: 'neo4j'
        Message: []
```

`neo4j` returns a `Neo4jConnect` object with these properties:

- **URL** — The Neo4j database web location
- **UserName** — The user name used to connect to the database
- **Message** — Any database connection error messages

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful Neo4j database connection.

```
neo4jconn.Message
```

```
ans =
    []
```

Retrieve all node labels using the Neo4j database connection `neo4jconn`. The cell array `nlabels` contains a character vector for the one node label in the Neo4j database.

```
nlabels = nodeLabels(neo4jconn)
```

```
nlabels = 1x1 cell array  
    {'Person'}
```

Close the database connection.

```
close(neo4jconn)
```

Connect to Neo4j Database Using Bolt Protocol

Create a Neo4j® database connection using the Database Toolbox™ Interface for Neo4j Bolt Protocol. Use a Bolt protocol URL to connect to the Neo4j database.

Create a Neo4j database connection using the Bolt protocol URL `bolt://localhost:7687/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'bolt://localhost:7687/db/data';  
username = 'neo4j';  
password = 'matlab';  
  
neo4jconn = neo4j(url,username,password)  
  
neo4jconn =  
    Neo4jConnect with properties:  
  
        URL: 'bolt://localhost:7687/db/data'  
    UserName: 'neo4j'  
    Message: []
```

`neo4j` returns a `Neo4jConnect` object with these properties:

- `URL` — The Neo4j database web location
- `UserName` — The user name used to connect to the database
- `Message` — Any database connection error messages

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful Neo4j database connection.

```
neo4jconn.Message
```

```
ans =  
  
    []
```

Retrieve all node labels using the Neo4j database connection `neo4jconn`. The cell array `nlabels` contains a character vector for the one node label in the Neo4j database.

```
nlabels = nodeLabels(neo4jconn)
```

```
nlabels = 1x1 cell array
    {'Person'}
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

url — Neo4j database connection URL

character vector | string scalar

Neo4j database connection URL that contains the server, port number, and web location of the Neo4j database, specified as a character vector or string scalar.

If you specify a URL that starts with the `http://` protocol identifier, then the `neo4j` function uses the REST API to connect to the Neo4j database.

If you specify a Bolt database connection URL that starts with the `bolt://` protocol identifier, that is, you use the Database Toolbox Interface for Neo4j Bolt Protocol, then the `neo4j` function creates a Bolt connection instead.

Note Ensure that you use the correct port number in the Neo4j database connection URL when you use the Bolt protocol. The default port number 7687 for the Bolt protocol is different from the default port numbers 7474 and 7473 for the HTTP and HTTPS protocols, respectively.

If you specify any other protocol identifier, then the `neo4j` function uses the REST API to create the database connection.

Example: `http://localhost:7474/db/data` specifies using the HTTP protocol where `localhost` is the server, 7474 is the port number, and `/db/data` is the web location of the database.

Example: `bolt://localhost:7687/db/data` specifies using the Bolt protocol where `localhost` is the server, 7687 is the port number, and `/db/data` is the web location of the database.

Data Types: `char` | `string`

username — User name

character vector | string scalar

User name for accessing the Neo4j database, specified as a character vector or string scalar. If no database authentication is required, specify an empty character vector.

Data Types: `char` | `string`

password — Password

character vector | string scalar

Password for accessing the Neo4j database, specified as a character vector or string scalar. If no database authentication is required, specify an empty character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

neo4j conn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, returned as a Neo4jConnect object.

Limitations

- The REST API and Database Toolbox Interface for Neo4j Bolt Protocol do not support Neo4j database versions 4.0 and later.

Version History

Introduced in R2016b

See Also

`nodeLabels` | `relationTypes` | `propertyKeys` | `close` | `searchGraph`

Topics

“Determine Dependencies of Services in Network” on page 9-22

“Find Shortest Path Between People in Social Neighborhood” on page 9-27

“Find Friends of Friends in Social Neighborhood” on page 9-32

“Explore Graph Database Structure” on page 9-2

“Search Graph Database” on page 9-9

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Error Messages for Neo4j Database Interfaces” on page 9-20

close

Package: database.neo4j.http

Close Neo4j database connection

Syntax

```
close(neo4jconn)
```

Description

`close(neo4jconn)` closes the Neo4j database connection.

Examples

Connect to Neo4j Database

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password)

neo4jconn =
  Neo4jConnect with properties:
      URL: 'http://localhost:7474/db/data/'
  UserName: 'neo4j'
  Message: []
```

`neo4j` returns a `Neo4jConnect` object with these properties:

- **URL** — The Neo4j database web location
- **UserName** — The user name used to connect to the database
- **Message** — Any database connection error messages

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful Neo4j database connection.

```
neo4jconn.Message
```

```
ans =
  []
```

Retrieve all node labels using the Neo4j database connection `neo4jconn`. The cell array `nlabels` contains a character vector for the one node label in the Neo4j database.

```
nlabels = nodeLabels(neo4jconn)
```

```
nlabels = 1x1 cell array  
    {'Person'}
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function `neo4j`.

Version History

Introduced in R2019a

See Also

`neo4j`

Topics

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Search Graph Database” on page 9-9

“Determine Dependencies of Services in Network” on page 9-22

“Find Shortest Path Between People in Social Neighborhood” on page 9-27

“Find Friends of Friends in Social Neighborhood” on page 9-32

“Explore Graph Database Structure” on page 9-2

“Error Messages for Neo4j Database Interfaces” on page 9-20

nodeLabels

Package: database.neo4j.http

All node labels in Neo4j database

Syntax

```
nlabels = nodeLabels(neo4jconn)
```

Description

`nlabels = nodeLabels(neo4jconn)` returns all node labels in the Neo4j database using the Neo4j database connection `neo4jconn`. You can retrieve the entire graph or search for a subgraph using the node labels. To search the graph database for relationship types instead, see `relationTypes`.

Examples

Retrieve All Node Labels

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';  
username = 'neo4j';  
password = 'matlab';  
  
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Retrieve all node labels using the Neo4j database connection `neo4jconn`. The cell array `nlabels` contains a character vector for the one node label in the Neo4j database.

```
nlabels = nodeLabels(neo4jconn)
```

```
nlabels = 1x1 cell array  
    {'Person'}
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

Output Arguments

nlabels — Node labels

cell array of character vectors

Node labels in the Neo4j database, returned as a cell array of character vectors. Each character vector denotes a node label.

Version History

Introduced in R2016b

See Also

neo4j | relationTypes | propertyKeys | close

Topics

“Explore Graph Database Structure” on page 9-2

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Search Graph Database” on page 9-9

“Error Messages for Neo4j Database Interfaces” on page 9-20

relationTypes

Package: database.neo4j.http

All relationship types in Neo4j database

Syntax

```
rtypes = relationTypes(neo4jconn)
```

Description

`rtypes = relationTypes(neo4jconn)` returns all relationship types in the Neo4j database using the Neo4j database connection `neo4jconn`. You can retrieve the entire graph or search for a subgraph using the relationship types. To search the graph database for node labels instead, see `nodeLabels`.

Examples

Retrieve All Relationship Types

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';  
username = 'neo4j';  
password = 'matlab';  
  
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Retrieve all relationship types using the Neo4j database connection `neo4jconn`. The cell array `rtypes` contains a character vector for the one relationship type in the Neo4j database.

```
rtypes = relationTypes(neo4jconn)  
  
rtypes = 1x1 cell array  
    {'knows'}
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

Output Arguments

rtypes — Relationship types

cell array of character vectors

Relationship types in the Neo4j database, returned as a cell array of character vectors. Each character vector denotes a relationship type.

Version History

Introduced in R2016b

See Also

neo4j | nodeLabels | propertyKeys | close

Topics

“Explore Graph Database Structure” on page 9-2

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Search Graph Database” on page 9-9

“Error Messages for Neo4j Database Interfaces” on page 9-20

propertyKeys

Package: database.neo4j.http

All property keys in Neo4j database

Syntax

```
propkeys = propertyKeys(neo4jconn)
```

Description

`propkeys = propertyKeys(neo4jconn)` returns all property keys in the Neo4j database using the Neo4j database connection `neo4jconn`.

Examples

Retrieve All Property Keys

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Retrieve all property keys using the Neo4j database connection `neo4jconn`. The cell array `propkeys` contains a character vector for the one property key in the Neo4j database.

```
propkeys = propertyKeys(neo4jconn)
```

```
propkeys = 15x1 cell array
    {'Weight' }
    {'EndNodes' }
    {'Address' }
    {'Department' }
    {'title' }
    {'name' }
    {'Name' }
    {'EndDate' }
    {'Description' }
```

```
{'StartDate' }  
{'property' }  
{'Date' }  
{'Project' }  
{'Location' }  
{'Title' }
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

Output Arguments

propkeys — Property keys

cell array of character vectors

Property keys in the Neo4j database, returned as a cell array of character vectors. Each character vector denotes a property key.

Version History

Introduced in R2016b

See Also

neo4j | nodeLabels | relationTypes | close

Topics

“Explore Graph Database Structure” on page 9-2

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Error Messages for Neo4j Database Interfaces” on page 9-20

searchNodeByID

Package: database.neo4j.http

Search Neo4j database nodes by node identifier

Syntax

```
nodeinfo = searchNodeByID(neo4jconn,nodeid)
```

Description

`nodeinfo = searchNodeByID(neo4jconn,nodeid)` creates the `Neo4jNode` object using the Neo4j database connection `neo4jconn` and the node identifier `nodeid`.

Examples

Search Neo4j Database by Node Identifier

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Search the database for the node with the node identifier 2 by using the Neo4j database connection `neo4jconn`.

```
nodeid = 2;
```

```
nodeinfo = searchNodeByID(neo4jconn,nodeid)
```

```
nodeinfo =
```

```
 Neo4jNode with properties:
```

```
   NodeID: 2
   NodeData: [1x1 struct]
   NodeLabels: 'Person'
```

`nodeinfo` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

Access the property keys and values of the node using the property `NodeData`.

```
nodeinfo.NodeData
```

```
ans = struct with fields:  
  name: 'User2'
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function `neo4j`.

nodeid — Node identifier

numeric scalar | numeric vector

Node identifier of a Neo4j database node, specified as a numeric scalar for one node in the Neo4j database, or a numeric vector for multiple nodes. If a node identifier is unknown, search for nodes using `searchNode` and search for relationships using `searchRelation`.

Data Types: `double`

Output Arguments

nodeinfo — Node information

Neo4jNode object

Node information for one node in the Neo4j database, returned as a Neo4jNode object. You can use this node as the origin node for searching the Neo4j database.

Version History

Introduced in R2016b

See Also

`neo4j` | `nodeRelationTypes` | `nodeDegree` | `close`

Topics

“Explore Graph Database Structure” on page 9-2

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Search Graph Database” on page 9-9

searchNode

Package: database.neo4j.http

Search Neo4j database nodes by label or by property key and value

Syntax

```
nodeinfo = searchNode(neo4jconn,nlabel)
nodeinfo = searchNode(neo4jconn,nlabel,Name,Value)
```

Description

`nodeinfo = searchNode(neo4jconn,nlabel)` returns node information for nodes with a specific node label using the Neo4j database connection `neo4jconn`.

`nodeinfo = searchNode(neo4jconn,nlabel,Name,Value)` narrows the search for nodes with additional options specified by the `Name,Value` pair arguments.

Examples

Search Nodes by Node Label

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Search the database for nodes that have node label `Person` using the Neo4j database connection `neo4jconn`.

```
nlabel = 'Person';

nodeinfo = searchNode(neo4jconn,nlabel)
```

```
nodeinfo=7×3 table
      NodeLabels      NodeData      NodeObject
```

```
0      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
1      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
2      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
3      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
4      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
5      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
9      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
```

`nodeinfo` is a table that contains information for each database node:

- Each row name is a node identifier.
- Variable `NodeLabels` is the node label.
- Variable `NodeData` is the node information.
- Variable `NodeObject` is the `Neo4jNode` object.

Access the node information for the first node in the table. The structure contains one property key and value.

```
node = nodeinfo.NodeData(1);
node{1}

ans = struct with fields:
    name: 'User1'
```

Access the node information using the row name as an index. The structure contains one property key and value.

```
nodeinfo.NodeData{'0'}

ans = struct with fields:
    name: 'User1'
```

Find the node degree for the first database node in the table. Specify outgoing relationships. There are two outgoing relationships from the first node in the table with relationship type `knows`.

```
degree = nodeDegree(nodeinfo.NodeObject(1), 'out')

degree = struct with fields:
    knows: 2
```

Close the database connection.

```
close(neo4jconn)
```

Search Nodes by Property Key and Value

Create a `Neo4j@` database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
```

```
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Search the database for nodes that have node label `Person` using the Neo4j database connection `neo4jconn`. Filter the results further by the property key and value for a specific person named `User2`. The `nodeinfo` output argument is a `Neo4jNode` object that contains node information.

```
nlabel = 'Person';
```

```
nodeinfo = searchNode(neo4jconn,nlabel,'PropertyKey','name', ...
    'PropertyValue','User2')
```

```
nodeinfo =
```

```
Neo4jNode with properties:
```

```
    NodeID: 2
    NodeData: [1x1 struct]
    NodeLabels: 'Person'
```

Access the node information. The structure contains a property key and value for `User2`.

```
nodeinfo.NodeData
```

```
ans = struct with fields:
```

```
    name: 'User2'
```

Find the node degree of the outgoing relationships. There is one outgoing relationship type `knows` for `User2`.

```
degree = nodeDegree(nodeinfo,'out')
```

```
degree = struct with fields:
```

```
    knows: 1
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a `Neo4jConnect` object created with the function `neo4j`.

nLabel — Neo4j database node label

character vector | string scalar

Neo4j database node label, specified as a character vector or string scalar.

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `nodeinfo =`

```
searchNode(neo4jconn, 'Person', 'PropertyKey', 'name', 'PropertyValue', 'User2');
```

PropertyKey — Property key

character vector | string scalar

Property key, specified as a comma-separated pair consisting of `'PropertyKey'` and a character vector or string scalar. A property key must have a corresponding property value. To specify the property value, use the name-value pair argument `'PropertyValue'`.

Example: `'PropertyKey', 'name'`

Data Types: char | string

PropertyValue — Property value

character vector | string scalar

Property value, specified as a comma-separated pair consisting of `'PropertyValue'` and a character vector or string scalar. A property value must have a corresponding property key. To specify the property key, use the name-value pair argument `'PropertyKey'`.

Example: `'PropertyValue', 'User1'`

Data Types: char | string

Output Arguments**nodeinfo — Node information**

Neo4jNode object | table

Node information in the Neo4j database, returned as a `Neo4jNode` object for one node or as a table for multiple nodes.

For multiple nodes, the table contains these variables:

- `NodeLabels` — Cell array of character vectors that contains the node labels for each database node
- `NodeData` — Cell array of structures that contains node information such as property keys
- `NodeObject` — `Neo4jNode` object for each database node

The row names of the table are Neo4j node identifiers of each database node.

Version History

Introduced in R2016b

See Also

neo4j | searchNodeByID | searchRelation | nodeRelationTypes | nodeDegree | searchGraph | close

Topics

“Explore Graph Database Structure” on page 9-2

“Search Graph Database” on page 9-9

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Error Messages for Neo4j Database Interfaces” on page 9-20

searchRelation

Package: database.neo4j.http

Search relationships for Neo4j database node

Syntax

```
relnfo = searchRelation(neo4jconn,nodeinfo,direction)
relnfo = searchRelation(neo4jconn,nodeinfo,direction,Name,Value)
```

Description

`relnfo = searchRelation(neo4jconn,nodeinfo,direction)` returns relationship information for the origin node `nodeinfo` and relationship direction using a Neo4j database connection. The search starts from the origin node. To find an origin node, use `searchNode` or `searchNodeByID`.

`relnfo = searchRelation(neo4jconn,nodeinfo,direction,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'RelationTypes', {'works with'}` returns information for relationships that have the type `works with`.

Examples

Search for Incoming Relationship

Search for information about a relationship in a Neo4j® database and display the information.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from `User1` through `User7`. Each relationship has the type `knows`.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Retrieve the origin node `nodeinfo` using the Neo4j database connection and the node identifier 3.


```

nodeid = 3;
nodeinfo = searchNodeByID(neo4jconn,nodeid);

Search for incoming relationships using the Neo4j database connection and the origin node
nodeinfo.

direction = 'in';

relinfo = searchRelation(neo4jconn,nodeinfo,direction)

relinfo = struct with fields:
    Origin: 3
    Nodes: [2x3 table]
    Relations: [1x5 table]

```

`relinfo` is a structure that contains the results of the search:

- **Origin** — The node identifier for the specified origin node
- **Nodes** — A table containing all start and end nodes for each matched relationship
- **Relations** — A table containing all matched relationships

Access the table of nodes.

`relinfo.Nodes`

```

ans=2x3 table
      NodeLabels      NodeData      NodeObject
      _____      _____      _____
      1      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      3      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]

```

Access the table of relationships.

`relinfo.Relations`

```

ans=1x5 table
      StartNodeID      RelationType      EndNodeID      RelationData      RelationObject
      _____      _____      _____      _____      _____
      3      1      'knows'      3      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]

```

Close the database connection.

```
close(neo4jconn)
```

Search Relationships by Type and Distance

Search for information about relationships in a Neo4j® database and display the information. Specify the relationship type and distance to search.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type knows.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Retrieve the origin node `nodeinfo` using the Neo4j database connection and the node identifier 3.

```
nodeid = 3;

nodeinfo = searchNodeByID(neo4jconn,nodeid);
```

Search for incoming relationships using the Neo4j database connection and the origin node `nodeinfo`. Refine the search by filtering for the relationship type `knows` and for nodes at a distance of two or fewer.

```
direction = 'in';
reltypes = {'knows'};

relinfo = searchRelation(neo4jconn,nodeinfo,direction, ...
    'RelationTypes',reltypes,'Distance',2)

relinfo = struct with fields:
    Origin: 3
    Nodes: [4x3 table]
    Relations: [3x5 table]
```

`relinfo` is a structure that contains the results of the search:

- `Origin` — The node identifier for the specified origin node
- `Nodes` — A table containing all start and end nodes for each matched relationship
- `Relations` — A table containing all matched relationships

Access the table of nodes.

```
relinfo.Nodes
```

```
ans=4x3 table
```

```
NodeLabels
```

```
NodeData
```

```
NodeObject
```

```

0      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
1      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
2      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
3      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]

```

Access the table of relationships.

```
reinfo.Relations
```

```
ans=3x5 table
```

	StartNodeID	RelationType	EndNodeID	RelationData	RelationObject
3	1	'knows'	3	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
2	2	'knows'	1	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
1	0	'knows'	1	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]

Close the database connection.

```
close(neo4jconn)
```

Return Relationship Information as Directed Graph

Search for information about outgoing relationships in a Neo4j® database. Return the information as a directed graph and display the edges and nodes of the graph.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type knows.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Retrieve the origin node `nodeinfo` using the Neo4j database connection and the node identifier 3.

```
nodeid = 3;
nodeinfo = searchNodeByID(neo4jconn,nodeid);
```

Search for outgoing relationships using the Neo4j database connection and the origin node `nodeinfo`. Return relationship information as a directed graph by using the `'DataReturnFormat'` name-value pair argument with the value `'digraph'`.

```
direction = 'out';
relinfo = searchRelation(neo4jconn,nodeinfo,direction, ...
    'DataReturnFormat','digraph')

relinfo =
    digraph with properties:

        Edges: [2x3 table]
        Nodes: [3x3 table]
```

Display the edges of the directed graph.

```
relinfo.Edges
```

```
ans=2x3 table
      EndNodes      RelationType      RelationData
      _____      _____      _____
      {'3'}      {'4'}      {'knows'}      {1x1 struct}
      {'3'}      {'5'}      {'knows'}      {1x1 struct}
```

Display the nodes of the directed graph.

```
relinfo.Nodes
```

```
ans=3x3 table
      Name      NodeLabels      NodeData
      _____      _____      _____
      {'3'}      {'Person'}      {1x1 struct}
      {'4'}      {'Person'}      {1x1 struct}
      {'5'}      {'Person'}      {1x1 struct}
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function `neo4j`.

nodeinfo — Origin node information

Neo4jNode object | numeric scalar

Origin node information, specified as a Neo4jNode object or numeric scalar that denotes a node identifier.

Data Types: double

direction — Relationship direction

'in' | 'out'

Relationship direction, specified as either 'in' for an incoming relationship or 'out' for an outgoing relationship. The relationships are associated with the specified origin node.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . ,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `reinfo = searchRelation(neo4jconn,nodeinfo,'in','RelationTypes',{'knows'},'Distance',2)` returns the relationship information for the incoming relationships, which have the relationship type `knows` and are two or fewer nodes away from the origin node.

RelationTypes — Relationship types

character vector | string scalar | cell array of character vectors | string array

Relationship types, specified as a comma-separated pair consisting of 'RelationTypes' and a character vector, string scalar, cell array of character vectors, or string array. To search for relationships using only one relationship type, use a character vector or string scalar. To search for relationships using numerous relationship types, use a cell array of character vectors or string array.

Example: 'RelationTypes',{'knows'}

Data Types: char | cell | string

Distance — Node distance

numeric scalar

Node distance, specified as a comma-separated pair consisting of 'Distance' and a positive numeric scalar. For example, if the node distance is three, `searchRelation` returns information for nodes that are three or fewer nodes away from the origin node `nodeinfo`.

Example: 'Distance',3

Data Types: double

DataReturnFormat — Data return format

'struct' (default) | 'digraph'

Data return format, specified as the comma-separated pair consisting of 'DataReturnFormat' and the value 'struct' for a structure or 'digraph' for a digraph object. Specify this argument to return relationship information as a digraph object.

Output Arguments

reinfo — Relationship information

structure

Relationship information in the Neo4j database that matches the search criteria from the origin node `nodeinfo`, returned as a structure with these fields.

Field	Description
Origin	Node identifier of the origin node <code>nodeinfo</code> .
Nodes	Table that contains node information for each node in the <code>Relations</code> table. The <code>Nodes</code> table contains the following information: <ul style="list-style-type: none"> <code>NodeLabels</code> — Character vector that denotes the node label for each matched database node <code>NodeData</code> — Structure array that contains node information such as property keys for each matched database node <code>NodeObject</code> — <code>Neo4jNode</code> object that represents each matched database node The row names in the table are Neo4j node identifiers of the matched database nodes.
Relations	Table that contains relationship information for the nodes in the <code>Nodes</code> table. The <code>Relations</code> table contains the following information: <ul style="list-style-type: none"> <code>StartNodeID</code> — Node identifier for the start node for each matched relationship <code>RelationType</code> — Character vector that denotes the relationship type for each matched relationship <code>EndNodeID</code> — Node identifier for the end node for each matched relationship <code>RelationData</code> — Structure array that contains property keys associated with each matched relationship <code>RelationObject</code> — <code>Neo4jRelation</code> object that represents each matched relationship The row names in the table are Neo4j relationship identifiers.

Note When you use the `'DataReturnFormat'` name-value pair argument with the value `'digraph'`, the `searchRelation` function returns relationship information in a `digraph` object. The resulting `digraph` object contains the same data as the `digraph` object created when you execute the `neo4jStruct2Digraph` function using the `relinfo` output argument.

Version History

Introduced in R2016b

See Also

`Neo4jRelation` | `neo4j` | `searchNode` | `searchNodeByID` | `searchGraph` | `searchRelationByID` | `close`

Topics

“Explore Graph Database Structure” on page 9-2

“Search Graph Database” on page 9-9

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Error Messages for Neo4j Database Interfaces” on page 9-20

searchGraph

Package: database.neo4j.http

Search for subgraph or entire graph in Neo4j database

Syntax

```
graphinfo = searchGraph(neo4jconn,criteria)
graphinfo = searchGraph(neo4jconn,criteria,'DataReturnFormat','digraph')
```

Description

`graphinfo = searchGraph(neo4jconn,criteria)` returns graph information based on the search criteria using a Neo4j database connection. You can search for a subgraph or the entire graph.

`graphinfo = searchGraph(neo4jconn,criteria,'DataReturnFormat','digraph')` returns graph information as a digraph object.

Examples

Search Graph by Node Labels

Search for graph information in a Neo4j® database by using node labels and display the information.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Search the graph for all nodes with the label `'Person'` using the Neo4j database connection.

```
nlabel = {'Person'};

graphinfo = searchGraph(neo4jconn,nlabel)
```

```
graphinfo = struct with fields:
    Nodes: [7×3 table]
    Relations: [8×5 table]
```

`graphinfo` is a structure that contains the results of the search:

- All start and end nodes that denote each matched relationship
- All matched relationships

Access the table of nodes.

```
graphinfo.Nodes
```

```
ans=7x3 table
      NodeLabels      NodeData      NodeObject
      _____      _____      _____
      0      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      1      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      2      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      3      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      4      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      5      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      9      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
```

Access property keys for the first node.

```
graphinfo.Nodes.NodeData{1}
```

```
ans = struct with fields:
      name: 'User1'
```

Access the table of relationships.

```
graphinfo.Relations
```

```
ans=8x5 table
      StartNodeID      RelationType      EndNodeID      RelationData      RelationObject
      _____      _____      _____      _____      _____
      1      0      'knows'      1      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      0      0      'knows'      2      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      3      1      'knows'      3      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      2      2      'knows'      1      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      5      3      'knows'      4      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      4      3      'knows'      5      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      6      5      'knows'      4      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      8      5      'knows'      9      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
```

Access property keys for the first relationship. The first relationship has no property keys.

```
graphinfo.Relations.RelationData{1}
```

```
ans = struct with no fields.
```

Search the graph for all node labels in the database.


```
allnodes = nodeLabels(neo4jconn);
graphinfo = searchGraph(neo4jconn,allnodes);
Close the database connection.
close(neo4jconn)
```

Search Graph by Relationships

Search for graph information in a Neo4j® database by using the relationship type and display the information.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Search the graph for the relationship type `'knows'` using the Neo4j database connection.

```
reltype = {'knows'};
graphinfo = searchGraph(neo4jconn,reltype)
```

```
graphinfo = struct with fields:
    Nodes: [7×3 table]
    Relations: [8×5 table]
```

`graphinfo` is a structure that contains the results of the search:

- All start and end nodes that denote each matched relationship
- All matched relationships

Access the table of nodes.

```
graphinfo.Nodes
```

```
ans=7×3 table
    NodeLabels    NodeData    NodeObject
    _____    _____    _____
    0    'Person'    [1×1 struct]    [1×1 database.neo4j.http.Neo4jNode]
```

```

2      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
1      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
3      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
5      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
4      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
9      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]

```

Access the table of relationships.

```
graphinfo.Relations
```

```
ans=8x5 table
      StartNodeID      RelationType      EndNodeID      RelationData      RelationObject
      _____      _____      _____      _____      _____
0      0      'knows'      2      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
1      0      'knows'      1      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
2      2      'knows'      1      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
3      1      'knows'      3      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
4      3      'knows'      5      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
5      3      'knows'      4      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
6      5      'knows'      4      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
8      5      'knows'      9      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]

```

Search the graph for all relationship types in the database.

```

allreltypes = relationTypes(neo4jconn);
graphinfo = searchGraph(neo4jconn,allreltypes);
Close the database connection.
close(neo4jconn)

```

Return Graph Information as Directed Graph

Search for graph information in a Neo4j® database by using node labels. Return the information as a directed graph and display the edges and nodes of the graph.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type knows.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```

url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);

```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Search the graph for all nodes with the node label `Person` using the Neo4j database connection. Return graph information as a directed graph by using the `'DataReturnFormat'` name-value pair argument with the value `'digraph'`.

```
nlabel = "Person";
graphinfo = searchGraph(neo4jconn,nlabel, ...
    'DataReturnFormat','digraph');
```

Display the edges of the directed graph.

```
graphinfo.Edges
```

```
ans=8x3 table
```

EndNodes	RelationType	RelationData
{'0'}	{'1'}	{'knows'}
{'0'}	{'2'}	{'knows'}
{'1'}	{'3'}	{'knows'}
{'2'}	{'1'}	{'knows'}
{'3'}	{'4'}	{'knows'}
{'3'}	{'5'}	{'knows'}
{'5'}	{'4'}	{'knows'}
{'5'}	{'9'}	{'knows'}

Display the nodes of the directed graph.

```
graphinfo.Nodes
```

```
ans=7x3 table
```

Name	NodeLabels	NodeData
{'0'}	{'Person'}	{'1x1 struct'}
{'1'}	{'Person'}	{'1x1 struct'}
{'2'}	{'Person'}	{'1x1 struct'}
{'3'}	{'Person'}	{'1x1 struct'}
{'4'}	{'Person'}	{'1x1 struct'}
{'5'}	{'Person'}	{'1x1 struct'}
{'9'}	{'Person'}	{'1x1 struct'}

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a `Neo4jConnect` object created with the function `neo4j`.

criteria — Search criteria

cell array of character vectors | string array

Search criteria, specified as a cell array of character vectors or string array. To search by nodes, specify one or more node labels as character vectors in the cell array. To search by relationships, specify one or more relationship types as character vectors in the cell array. Or, specify a string array for multiple node labels or relationship types.

Data Types: `cell` | `string`

Output Arguments

graphinfo — Graph information

structure

Graph information in the Neo4j database that matches the search criteria, returned as a structure with these fields.

Field	Description
Nodes	<p>Table that contains node information for each node in the <code>Relations</code> table. The <code>Nodes</code> table contains the following fields:</p> <ul style="list-style-type: none"> <code>NodeLabels</code> — Character vector that denotes the node label for each matched database node <code>NodeData</code> — Structure array that contains node information such as property keys for each matched database node <code>NodeObject</code> — <code>Neo4jNode</code> object for each matched database node <p>The row names in the table are Neo4j node identifiers of the matched database nodes.</p> <p>If <code>criteria</code> contains node labels, the output is automatically sorted by <code>StartNodeID</code> and <code>EndNodeID</code>.</p>
Relations	<p>Table that contains relationship information for the nodes in the <code>Nodes</code> table. The <code>Relations</code> table contains the following fields:</p> <ul style="list-style-type: none"> <code>StartNodeID</code> — Node identifier for the start node for each matched relationship <code>RelationType</code> — Character vector that denotes the relationship type for each matched relationship <code>EndNodeID</code> — Node identifier for the end node for each matched relationship <code>RelationData</code> — Structure array that contains property keys associated with each matched relationship <code>RelationObject</code> — <code>Neo4jRelation</code> object that represents each matched relationship <p>The row names in the table are Neo4j relationship identifiers.</p> <p>If <code>criteria</code> contains relationship types, the output is automatically sorted by <code>RelationID</code>.</p>

Note When you use the `'DataReturnFormat'` name-value pair argument with the value `'digraph'`, the `searchGraph` function returns graph information in a `digraph` object. The resulting `digraph` object contains the same data as the `digraph` object created when you execute the `neo4jStruct2Digraph` function using the `graphinfo` output argument.

Version History

Introduced in R2016b

See Also

neo4j | searchNode | searchNodeByID | searchRelation | relationTypes | nodeLabels | close

Topics

“Determine Dependencies of Services in Network” on page 9-22

“Find Shortest Path Between People in Social Neighborhood” on page 9-27

“Find Friends of Friends in Social Neighborhood” on page 9-32

“Search Graph Database” on page 9-9

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

executeCypher

Package: database.neo4j.http

Execute Cypher query on Neo4j database

Syntax

```
results = executeCypher(neo4jconn,query)
```

Description

`results = executeCypher(neo4jconn,query)` returns data from the Neo4j database using the Neo4j database connection `neo4jconn` and a Cypher query. You can execute a Cypher query on the Neo4j database using the Cypher Query Language.

Examples

Execute Cypher Query in Neo4j Database

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';  
username = 'neo4j';  
password = 'matlab';  
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Create the Cypher® query that searches for the names of all nodes with the node label `Person`.

```
query = 'MATCH (node:Person) RETURN node.name';
```

Execute the query and display the results using the Neo4j database connection `neo4jconn`.

```
results = executeCypher(neo4jconn,query)
```

```
results=7×1 table  
node_name  
-----  
 'User1'  
 'User3'  
 'User2'
```

```
'User4'
'User5'
'User6'
'User7'
```

`results` is a table that contains the column `node_name`. This column has the names of each node in the Neo4j database.

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function `neo4j`.

query — Cypher query

character vector | string scalar

Cypher query, specified as a character vector or string scalar.

Example: `'MATCH (movie: Movie {title: ''The Matrix''}) RETURN movie.title, movie.studio'`

Data Types: `char` | `string`

Output Arguments

results — Cypher query results

table

Cypher query results, returned as a table. The columns in the table match the RETURN statement in the Cypher query.

Version History

Introduced in R2016b

See Also

`neo4j` | `searchGraph` | `searchNode` | `searchNodeByID` | `searchRelation` | `close`

Topics

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Search Graph Database” on page 9-9

“Error Messages for Neo4j Database Interfaces” on page 9-20

nodeRelationTypes

Package: database.neo4j.http

Associated relationship types for Neo4j database node

Syntax

```
nodereltypes = nodeRelationTypes(node,direction)
```

Description

`nodereltypes = nodeRelationTypes(node,direction)` returns the relationship types for the specified `Neo4jNode` object and direction.

Examples

Search Relationship Types for Node

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';  
username = 'neo4j';  
password = 'matlab';  
  
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Search the database for the node with node identifier 2 using the Neo4j database connection `neo4jconn`.

```
nodeid = 2;  
  
node = searchNodeByID(neo4jconn,nodeid);
```

Search for all incoming relationships for the node. `nodereltypes` returns a list of the relationship types.

```
nodereltypes = nodeRelationTypes(node,'in')  
  
nodereltypes = 1x1 cell array  
    {'knows'}
```


Close the database connection.

```
close(neo4jconn)
```

Input Arguments

node — Neo4j database node

Neo4jNode object

Neo4j database node, specified as a Neo4jNode object created using `searchNode` or `searchNodeByID`.

direction — Relationship direction

'in' | 'out'

Relationship direction, specified as either 'in' for an incoming relationship or 'out' for an outgoing relationship. The relationships are associated with the specified origin node.

Output Arguments

nodereltypes — Relationship types

cell array of character vectors

Relationship types, returned as a cell array of character vectors. The cell array contains one character vector for one relationship or multiple character vectors for multiple relationships.

Version History

Introduced in R2016b

See Also

`nodeDegree` | `searchNodeByID` | `searchNode` | `close`

Topics

“Explore Graph Database Structure” on page 9-2

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Search Graph Database” on page 9-9

nodeDegree

Package: database.neo4j.http

In-degree and out-degree for each associated relationship type for Neo4j database node

Syntax

```
degree = nodeDegree(node,direction)
```

Description

`degree = nodeDegree(node,direction)` returns the in- or out-degree for each relationship for the specified Neo4jNode object. `direction` specifies the relationship direction.

Examples

Search Node Degree for Node

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';  
username = 'neo4j';  
password = 'matlab';  
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j® connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Search the database for the node with node identifier 2 using the Neo4j® database connection `neo4jconn`.

```
nodeid = 2;  
node = searchNodeByID(neo4jconn,nodeid);
```

Search for the degree of all incoming relationships for the node. `degree` returns a structure with the in-degree for each relationship type.

```
degree = nodeDegree(node,'in')
```

```
degree = struct with fields:  
    knows: 1
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

node — Neo4j database node

Neo4jNode object

Neo4j database node, specified as a Neo4jNode object created using `searchNode` or `searchNodeByID`.

direction — Relationship direction

'in' | 'out'

Relationship direction, specified as either 'in' for an incoming relationship or 'out' for an outgoing relationship. The relationships are associated with the specified origin node.

Output Arguments

degree — In- or out-degree

structure

In- or out-degree, returned as a structure. Each field in the structure represents either incoming or outgoing relationship types. If there are no incoming or outgoing relationship types, the structure is empty.

Version History

Introduced in R2016b

See Also

`searchNodeByID` | `searchNode` | `nodeRelationTypes` | `close`

Topics

“Explore Graph Database Structure” on page 9-2

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

“Search Graph Database” on page 9-9

neo4jStruct2Digraph

Convert graph or relationship structure from Neo4j database to directed graph

Syntax

```
G = neo4jStruct2Digraph(s)
G = neo4jStruct2Digraph(s, 'NodeNames', nodenames)
```

Description

`G = neo4jStruct2Digraph(s)` creates a directed graph from the structure `s`. With the directed graph, run graph network analytics using MATLAB. For example, to visualize the graph, see “Graph Plotting and Customization”.

`G = neo4jStruct2Digraph(s, 'NodeNames', nodenames)` specifies names of the Neo4j database nodes in the directed graph.

Examples

Create Directed Graph Using Relationships

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url, username, password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Search for incoming relationships using the Neo4j database connection `neo4jconn` and origin node identifier `nodeid`.

```
nodeid = 1;
direction = 'in';
```

```
relinfo = searchRelation(neo4jconn, nodeid, direction);
```

Convert the relationship information into a directed graph. `G` is a `digraph` object that contains two tables for edges and nodes.

```
G = neo4jStruct2Digraph(relinfo)
```

```
G =
  digraph with properties:

    Edges: [2x3 table]
    Nodes: [3x3 table]
```

Access the table of edges.

G.Edges

```
ans=2x3 table
  EndNodes      RelationType      RelationData
  _____      _____      _____
  '0'    '1'      'knows'      [1x1 struct]
  '2'    '1'      'knows'      [1x1 struct]
```

Access the table of nodes.

G.Nodes

```
ans=3x3 table
  Name      NodeLabels      NodeData
  _____      _____      _____
  '0'      'Person'      [1x1 struct]
  '1'      'Person'      [1x1 struct]
  '2'      'Person'      [1x1 struct]
```

Find the shortest path between all nodes in G.

```
d = distances(G)
```

```
d = 3x3
    0     1     Inf
  Inf     0     Inf
  Inf     1     0
```

Close the database connection.

```
close(neo4jconn)
```

Create Directed Graph Using a Subgraph

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Search for a subgraph using the Neo4j database connection `neo4jconn` and node label `nlabel`.

```
nlabel = {'Person'};
```

```
graphinfo = searchGraph(neo4jconn,nlabel);
```

Convert the graph information into a directed graph. `G` is a `digraph` object that contains two tables for edges and nodes.

```
G = neo4jStruct2Digraph(graphinfo)
```

```
G =
```

```
 digraph with properties:
```

```
 Edges: [8×3 table]
```

```
 Nodes: [7×3 table]
```

Access the table of edges.

G.Edges

```
ans=8×3 table
```

EndNodes	RelationType	RelationData
'0' '1'	'knows'	[1×1 struct]
'0' '2'	'knows'	[1×1 struct]
'1' '3'	'knows'	[1×1 struct]
'2' '1'	'knows'	[1×1 struct]
'3' '4'	'knows'	[1×1 struct]
'3' '5'	'knows'	[1×1 struct]
'5' '4'	'knows'	[1×1 struct]
'5' '9'	'knows'	[1×1 struct]

Access the table of nodes.

G.Nodes

```
ans=7×3 table
```

Name	NodeLabels	NodeData
'0'	'Person'	[1×1 struct]
'1'	'Person'	[1×1 struct]
'2'	'Person'	[1×1 struct]
'3'	'Person'	[1×1 struct]
'4'	'Person'	[1×1 struct]
'5'	'Person'	[1×1 struct]

```
'g'      'Person'      [1x1 struct]
```

Find the shortest path between all nodes in G.

```
d = distances(G)
```

```
d = 7×7
```

```

    0     1     1     2     3     3     4
  Inf     0   Inf     1     2     2     3
  Inf     1     0     2     3     3     4
  Inf  Inf  Inf     0     1     1     2
  Inf  Inf  Inf  Inf     0   Inf  Inf
  Inf  Inf  Inf  Inf     1     0     1
  Inf  Inf  Inf  Inf  Inf  Inf     0

```

Close the database connection.

```
close(neo4jconn)
```

Create Directed Graph Using Node Names

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Search for a subgraph using the Neo4j database connection `neo4jconn` and node label `nlabel`.

```
nlabel = {'Person'};
```

```
graphinfo = searchGraph(neo4jconn,nlabel);
```

Convert the graph information into a directed graph using the node names in the subgraph. Convert node names into a cell array of character vectors `nodenames`. `G` is a `digraph` object that contains two tables for edges and nodes.

```
names = [graphinfo.Nodes.NodeData{:}];
nodenames = {names(:).name};
```

```
G = neo4jStruct2Digraph(graphinfo, 'NodeNames', nodenames)
```

```
G =
digraph with properties:

Edges: [8x3 table]
Nodes: [7x3 table]
```

Access the table of edges.

G.Edges

```
ans=8x3 table
      EndNodes      RelationType      RelationID
-----
'User1'  'User3'      'knows'          1
'User1'  'User2'      'knows'          0
'User3'  'User4'      'knows'          3
'User2'  'User3'      'knows'          2
'User4'  'User5'      'knows'          5
'User4'  'User6'      'knows'          4
'User6'  'User5'      'knows'          6
'User6'  'User7'      'knows'          8
```

Access the table of nodes.

G.Nodes

```
ans=7x3 table
      Name      NodeLabels      NodeData
-----
'User1'  'Person'      [1x1 struct]
'User3'  'Person'      [1x1 struct]
'User2'  'Person'      [1x1 struct]
'User4'  'Person'      [1x1 struct]
'User5'  'Person'      [1x1 struct]
'User6'  'Person'      [1x1 struct]
'User7'  'Person'      [1x1 struct]
```

Find the shortest path between all nodes in G.

```
d = distances(G)
```

```
d = 7x7
      0      1      1      2      3      3      4
Inf      0      Inf      1      2      2      3
Inf      1      0      2      3      3      4
Inf      Inf      Inf      0      1      1      2
Inf      Inf      Inf      Inf      0      Inf      Inf
Inf      Inf      Inf      Inf      1      0      1
Inf      Inf      Inf      Inf      Inf      Inf      0
```

Close the database connection.


```
close(neo4jconn)
```

Input Arguments

s — Graph or relationship information

structure

Graph or relationship information, specified as a structure returned by `searchGraph` or `searchRelation`.

Data Types: `struct`

nodenames — Node names

cell array of character vectors | string array

Node names in a Neo4j database, specified as a cell array of character vectors or string array.

Example: `["User6", "User7"]`

Data Types: `cell` | `string`

Output Arguments

G — Directed graph

digraph object

Directed graph, returned as a `digraph` object.

Version History

Introduced in R2016b

See Also

`neo4j` | `searchNode` | `searchGraph` | `searchRelation` | `distances` | `close`

Topics

“Determine Dependencies of Services in Network” on page 9-22

“Find Shortest Path Between People in Social Neighborhood” on page 9-27

“Find Friends of Friends in Social Neighborhood” on page 9-32

“Search Graph Database” on page 9-9

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

createNode

Package: database.neo4j.http

Create nodes in Neo4j database

Syntax

```
createNode(neo4jconn)
createNode(neo4jconn,Name,Value)
nodeinfo = createNode( ___ )
```

Description

`createNode(neo4jconn)` creates a single node without labels and properties by using a Neo4j database connection.

`createNode(neo4jconn,Name,Value)` creates a single node or multiple nodes and returns node information by specifying additional options using one or more name-value pair arguments. For example, 'Labels', 'Person' creates a node with the label Person.

`nodeinfo = createNode(___)` returns node information as a `Neo4jNode` object for one node, or as a table for multiple nodes, using any of the input argument combinations in the previous syntaxes.

Examples

Create Single Node

Create a single node in a Neo4j® database and display the contents of the node.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Create a single node in the database using the Neo4j database connection.

```
createNode(neo4jconn)
```

When you execute the `createNode` function without any arguments except the Neo4j database connection, the function creates a single node without labels and properties.

Close the database connection.

```
close(neo4jconn)
```

Create Single Node with Label

Create a single node with a label in a Neo4j® database and access the node.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Create a single node with a label in the database by using the Neo4j database connection. Use the `'Labels'` name-value pair argument to specify the label `Scientist`.

```
label = 'Scientist';
createNode(neo4jconn, 'Labels', label)
```

Search for the new node using the label `Scientist`.

```
nodeinfo = searchNode(neo4jconn, label)
```

```
nodeinfo =
    Neo4jNode with properties:
        NodeID: 6
        NodeData: [1x1 struct]
        NodeLabels: 'Scientist'
```

`nodeinfo` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

Close the database connection.

```
close(neo4jconn)
```

Create Two Nodes with Labels

Create two nodes with labels in a Neo4j® database. Access data in the nodes.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Create two nodes that represent two people in the database by using the Neo4j database connection. Use the `'Labels'` name-value pair argument to specify the node labels. One node has the label `Person`, and the other node has two labels, `Person` and `Employee`.

```
labels = {'Person'},{'Person','Employee'};
nodeinfo = createNode(neo4jconn,'Labels',labels)
```

```
nodeinfo=2x3 table
      NodeLabels      NodeData      NodeObject
      _____      _____      _____
      35      'Person'      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      36      {2x1 cell}      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
```

`nodeinfo` is a table with two rows, one for each person. The table contains these variables:

- Node label
- Node data
- Neo4jNode object

Access the `Neo4jNode` object for the first node.

```
data = nodeinfo.NodeObject(1)
```

```
data =
  Neo4jNode with properties:
```

```
      NodeID: 35
      NodeData: [1x1 struct]
```

```
NodeLabels: 'Person'
```

data is a Neo4jNode object with these properties:

- Node identifier
- Node data
- Node label

Access the node labels of both nodes.

```
nodeinfo.NodeObject(1).NodeLabels
```

```
ans =
'Person'
```

```
nodeinfo.NodeObject(2).NodeLabels
```

```
ans = 2x1 cell array
    {'Person' }
    {'Employee'}
```

Close the database connection.

```
close(neo4jconn)
```

Create Two Nodes with Labels and Properties

Create two nodes with labels and properties in a Neo4j® database. Access the data in the nodes.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type knows.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =

[]
```

Create a table with two rows that contain the names and job titles of two people (users).

```
props = table(["User8";"User9"],["Analyst";"Technician"], ...
  'VariableNames',{ 'Name', 'Title'});
```

Create two nodes that represent these two people in the database by using the Neo4j database connection. Use the 'Labels' name-value pair argument to specify the node labels Person and Employee. Then, use the 'Properties' name-value pair argument to specify the node properties using the props table.

```
labels = ["Person","Employee"];
nodeinfo = createNode(neo4jconn,'Labels',labels,'Properties',props)
```

```
nodeinfo=2x3 table
      NodeLabels      NodeData      NodeObject
      _____      _____      _____
      28      {2x1 cell}      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
      21      {2x1 cell}      [1x1 struct]      [1x1 database.neo4j.http.Neo4jNode]
```

nodeinfo is a table with two rows, one for each person. The table contains these variables:

- Node label
- Node data
- Neo4jNode object

Access the properties of the first node. This structure contains the properties of the node as fields and values.

```
nodeinfo.NodeData{1}
```

```
ans = struct with fields:
  Title: 'Analyst'
  Name: 'User8'
```

Access the Neo4jNode object for the first node.

```
data = nodeinfo.NodeObject(1)
```

```
data =
  Neo4jNode with properties:
    NodeID: 28
    NodeData: [1x1 struct]
    NodeLabels: {2x1 cell}
```

data is a Neo4jNode object with these properties:

- Node identifier
- Node data
- Node label

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: node = createNode(neo4jconn, 'Labels', 'Person', 'Properties', table(["User8"], ["Analyst"], 'VariableNames', {'Name', 'Title'})) creates a single node with the label Person and two properties, Name and Title, with their corresponding values, User8 and Analyst.

Labels — Node labels

character vector | string scalar | cell array of character vectors | string array | cell array of cell arrays
| cell array of string arrays

Node labels, specified as the comma-separated pair consisting of 'Labels' and a character vector, string scalar, cell array of character vectors, string array, cell array of cell arrays, or cell array of string arrays. To specify one node label, use a character vector or string scalar. For multiple node labels, use a cell array of character vectors or a string array. To create multiple nodes with different labels, use a cell array of cell arrays or a cell array of string arrays.

Note If you do not specify any labels, then the created node has no labels by default.

Example: 'Labels', 'Person'

Data Types: char | string | cell

Properties — Node properties

structure | structure array | table | cell array of structures

Node properties, specified as the comma-separated pair consisting of 'Properties' and a structure, structure array, table, or cell array of structures.

When you specify a structure, the createNode function converts each field and its corresponding value to a property and its corresponding value in the database node. The function also sets the NodeData property of the Neo4jNode object to this structure.

When you specify a table that contains one row, the createNode function converts each variable and its corresponding value to a property and its corresponding value in the database node. The function also converts the variables and their corresponding values to fields and their corresponding values in a structure. The function sets this structure to the NodeData property of the Neo4jNode object.

To create multiple nodes, specify a structure array or table with multiple rows.

To create multiple nodes with different properties, specify a cell array of structures.

Note If a property is missing its corresponding value, then the created node does not contain this property.

Data Types: `struct` | `table` | `cell`

Output Arguments

nodeinfo — Node information

Neo4jNode object | table

Node information in the Neo4j database, returned as a Neo4jNode object for one node or as a table for multiple nodes.

For multiple nodes, the table contains these variables:

- `NodeLabels` — Cell array of character vectors that contains the node labels for each database node
- `NodeData` — Cell array of structures that contains node information such as property keys
- `NodeObject` — Neo4jNode object for each database node

The row names of the table are Neo4j node identifiers of each database node.

Version History

Introduced in R2018a

See Also

`deleteNode` | `neo4j` | `updateNode` | `addNodeLabel` | `removeNodeLabel` | `setNodeProperty` | `removeNodeProperty` | `close`

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

createRelation

Package: database.neo4j.http

Create relationships between nodes in Neo4j database

Syntax

```
createRelation(neo4jconn, startnode, endnode, relationtype)
createRelation(neo4jconn, startnode, endnode, relationtype, 'Properties',
properties)
relationinfo = createRelation( ___ )
```

Description

`createRelation(neo4jconn, startnode, endnode, relationtype)` creates a single relationship or multiple relationships between the start nodes and end nodes with specified relationship types by using the Neo4j database connection.

`createRelation(neo4jconn, startnode, endnode, relationtype, 'Properties', properties)` specifies the properties of the new relationships.

`relationinfo = createRelation(___)` returns relationship information as a `Neo4jRelation` object or a table using any of the input argument combinations in the previous syntaxes.

Examples

Create Single Relationship Between Two Nodes

Create a single relationship between two nodes in a Neo4j® database and display the relationship.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url, username, password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Create two nodes in the database using the Neo4j database connection. Use the 'Labels' name-value pair argument to specify the label `Person` for each node.

```
label = 'Person';
startnode = createNode(neo4jconn, 'Labels', label);
endnode = createNode(neo4jconn, 'Labels', label);
```

Create a relationship between the two nodes using the Neo4j database connection. Specify the relationship type as `works with`.

```
relationtype = 'works with';
createRelation(neo4jconn, startnode, endnode, relationtype)
```

Search for the new relationship and display its relationship type.

```
direction = "out";
relnfo = searchRelation(neo4jconn, startnode, direction, ...
    'RelationTypes', relationtype, 'Distance', 2);
relnfo.Relations.RelationType
```

```
ans = 1x1 cell array
    {'works with'}
```

Close the database connection.

```
close(neo4jconn)
```

Create Single Relationship with Properties

Create a single relationship with properties between two nodes in a Neo4j® database and display the properties.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url, username, password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =

    []
```

Create two nodes in the database using the Neo4j database connection. Use the `'Labels'` name-value pair argument to specify the label `Person` for each node.

```
label = 'Person';
startnode = createNode(neo4jconn, 'Labels', label);
endnode = createNode(neo4jconn, 'Labels', label);
```

Create a relationship between the two nodes using the Neo4j database connection. The nodes represent two colleagues who started working together on a project named `Database` on September

1, 2017. Specify the relationship type as `works with`. Specify the project name and start date as properties of the relationship by using the `properties` structure.

```
relationtype = 'works with';
properties.Project = 'Database';
properties.StartDate = '09/01/2017';
createRelation(neo4jconn,startnode,endnode,relationtype, ...
    'Properties',properties)
```

Search for the new relationship and display its properties.

```
direction = "out";
relnfo = searchRelation(neo4jconn,startnode,direction, ...
    'RelationTypes',relationtype,'Distance',2);
relnfo.Relations.RelationData{1}
```

```
ans = struct with fields:
    StartDate: '09/01/2017'
    Project: 'Database'
```

Close the database connection.

```
close(neo4jconn)
```

Create Multiple Relationships

Create two relationships between nodes in a Neo4j® database and display the relationships.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from `User1` through `User7`. Each relationship has the type `knows`.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
    []
```

Search for the node with the label `Person` and the property key name set to the value `User7` by using the Neo4j database connection.

```
nlabel = 'Person';
user7 = searchNode(neo4jconn,nlabel,'PropertyKey','name', ...
    'PropertyValue','User7');
```

Create two nodes in the database using the Neo4j database connection. Use the 'Labels' name-value pair argument to specify the label Person for each node.

```
label = 'Person';
user8 = createNode(neo4jconn,'Labels',label);
user9 = createNode(neo4jconn,'Labels',label);
```

Create two relationships using the Neo4j database connection. Specify the relationship types as `works with` and `studies with`. The two relationships are:

- User8 works with User7
- User8 studies with User9

```
startnode = [user8,user8];
endnode = [user7,user9];
relationtype = {'works with','studies with'};
relationinfo = createRelation(neo4jconn,startnode,endnode,relationtype)
```

`relationinfo=2x5 table`

	StartNodeID	RelationType	EndNodeID	RelationData	RelationObject
9	6	'works with'	9	[1x1 struct]	[1x1 database.neo4j.http.f
7	6	'studies with'	7	[1x1 struct]	[1x1 database.neo4j.http.f

`relationinfo` is a table with these variables:

- Start node identifier
- Relationship type
- End node identifier
- Relationship data
- Neo4jRelation object

Display the Neo4jRelation object for the first relationship.

```
relation = relationinfo.RelationObject(1)
```

```
relation =
    Neo4jRelation with properties:

        RelationID: 9
        RelationData: [1x1 struct]
        StartNodeID: 6
        RelationType: 'works with'
        EndNodeID: 9
```

`relation` is a Neo4jRelation object with these properties:

- Relationship identifier

- Relationship data
- Start node identifier
- Relationship type
- End node identifier

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

startnode — Start node

numeric scalar | numeric vector | Neo4jNode object | Neo4jNode object array

Start node, specified as a numeric scalar, numeric vector, Neo4jNode object, or Neo4jNode object array. To specify database nodes using node identifiers, use a numeric scalar for one node or a numeric vector for multiple nodes. To specify database nodes as Neo4jNode objects, use the object for one node or an array of the objects for multiple nodes.

The number of start nodes must match the number of end nodes in endnode.

The number of start nodes equals the number of relationships created by the createRelation function in the Neo4j database.

Example: 8

endnode — End node

numeric scalar | numeric vector | Neo4jNode object | Neo4jNode object array

End node, specified as a numeric scalar, numeric vector, Neo4jNode object, or Neo4jNode object array. To specify database nodes using node identifiers, use a numeric scalar for one node or a numeric vector for multiple nodes. To specify database nodes as Neo4jNode objects, use the object for one node or an array of the objects for multiple nodes.

The number of end nodes must match the number of start nodes in startnode.

The number of end nodes equals the number of relationships created by the createRelation function in the Neo4j database.

Example: 9

relationtype — Relationship type

character vector | string scalar | cell array of character vectors | string array

Relationship type, specified as a character vector, string scalar, cell array of character vectors, or string array. To specify one relationship type, use a character vector or string scalar. To specify multiple relationship types, use a cell array of character vectors or a string array.

If you specify only one relationship type, all relationships must have the same type. Otherwise, the number of relationship types in this input argument must match the number of nodes in `startnode` and `endnode`.

Example: 'knows'

Data Types: `char` | `string` | `cell`

properties — Relationship properties

`structure` | `structure array` | `table` | `cell array of structures`

Relationship properties, specified as a structure, structure array, table, or cell array of structures.

When you specify a structure, the `createRelation` function converts each field and its corresponding value to a property and its corresponding value in the database relationship. When you specify a table with one row, the function converts each variable and its corresponding value to a property and its corresponding value in the database relationship.

The `createRelation` function also sets the `RelationObject` variable of the `relationinfo` output argument to the `Neo4jRelation` object, which contains the relationship information.

For multiple relationships, specify a structure array or table with multiple rows.

For multiple relationships with different properties, specify a cell array of structures.

Note If a property is missing its corresponding value, then the updated relationship does not contain this property.

Data Types: `struct` | `table` | `cell`

Output Arguments

relationinfo — Relationship information

`Neo4jRelation` object | `table`

Relationship information, returned as a `Neo4jRelation` object for one relationship or as a table for multiple relationships.

For multiple relationships, the table contains these variables:

- `StartNodeID` — Node identifier of the start node for each matched relationship
- `RelationType` — Character vector that denotes the relationship type for each matched relationship
- `EndNodeID` — Node identifier of the end node for each matched relationship
- `RelationData` — Structure array that contains property keys associated with each matched relationship
- `RelationObject` — `Neo4jRelation` object for each matched relationship

The row names in the table are `Neo4j` relationship identifiers.

Version History

Introduced in R2018a

See Also

neo4j | updateRelation | deleteRelation | setRelationProperty |
removeRelationProperty | close | createNode | searchNode | searchRelation

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

deleteNode

Package: database.neo4j.http

Delete nodes from Neo4j database

Syntax

```
deleteNode(neo4jconn,node)
deleteNode(neo4jconn,node,'DeleteRelations','true')
```

Description

`deleteNode(neo4jconn,node)` deletes a single node or multiple nodes using the Neo4j database connection. If a specified node has an associated relationship, this syntax throws an error.

`deleteNode(neo4jconn,node,'DeleteRelations','true')` deletes nodes and their associated relationships without throwing an error.

Examples

Delete Node

Create a single node in a Neo4j® database and delete the node.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Create a single node in the Neo4j database using the Neo4j database connection.

```
node = createNode(neo4jconn)
```

```
node =
```

```
    Neo4jNode with properties:
```

```
    NodeID: 7
    NodeData: [1x1 struct]
    NodeLabels: []
```


node is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node label

Delete the node using the Neo4j database connection.

```
deleteNode(neo4jconn,node)
```

Close the database connection.

```
close(neo4jconn)
```

Delete Node and Its Relationship

Create a single relationship between two nodes in a Neo4j® database. Then, delete one of the nodes and the relationship.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Create two nodes in the Neo4j database using the Neo4j database connection. Use the `'Labels'` name-value pair argument to specify the `Person` node label for each node.

```
label = 'Person';
startnode = createNode(neo4jconn,'Labels',label);
endnode = createNode(neo4jconn,'Labels',label);
```

Create a relationship between two nodes using the Neo4j database connection. Specify the relationship type as `works with`.

```
relationtype = 'works with';
relation = createRelation(neo4jconn,startnode,endnode,relationtype)
```

```
relation =
```

```
Neo4jRelation with properties:
```

```
    RelationID: 19
    RelationData: [1x1 struct]
```

```
StartNodeID: 14
RelationType: 'works with'
EndNodeID: 15
```

relation is a `Neo4jRelation` object with these properties:

- Relationship identifier
- Relationship data
- Start node identifier
- Relationship type
- End node identifier

Delete the first node and the associated relationship. Use this syntax to delete the node and relationship without throwing an error.

```
deleteNode(neo4jconn, startnode, 'DeleteRelations', true)
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

`Neo4jConnect` object

Neo4j database connection, specified as a `Neo4jConnect` object created with the function `neo4j`.

node — Node

`Neo4jNode` object | `Neo4jNode` object array | numeric scalar | numeric vector

Node in a Neo4j database, specified as a `Neo4jNode` object, `Neo4jNode` object array, numeric scalar, or a numeric vector. For one node, specify a `Neo4jNode` object or a numeric scalar. For multiple nodes, specify a `Neo4jNode` object array or a numeric vector.

The numeric scalar or vector must contain Neo4j database node identifiers.

Example: 15

Example: [2, 3, 4]

Version History

Introduced in R2018a

See Also

`neo4j` | `createNode` | `createRelation` | `updateNode` | `close`

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

deleteRelation

Package: database.neo4j.http

Delete relationships from Neo4j database

Syntax

```
deleteRelation(neo4jconn, relation)
```

Description

`deleteRelation(neo4jconn, relation)` deletes a single relationship or multiple relationships using the Neo4j database connection.

Examples

Delete Relationship from Neo4j Database

Create a single relationship between two nodes in a Neo4j® database. Then, delete the relationship and the corresponding nodes.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url, username, password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Create two nodes in the Neo4j database using the Neo4j database connection. Use the `'Labels'` name-value pair argument to specify the `Person` node label for each node.

```
label = 'Person';
startnode = createNode(neo4jconn, 'Labels', label);
endnode = createNode(neo4jconn, 'Labels', label);
```

Create a relationship between the two nodes using the Neo4j database connection. These nodes represent two colleagues who work together. Specify the relationship type as `works with`.

```
relationtype = 'works with';
relation = createRelation(neo4jconn, startnode, endnode, relationtype)
```

```
relation =  
  Neo4jRelation with properties:  
  
    RelationID: 17  
    RelationData: [1×1 struct]  
    StartNodeID: 52  
    RelationType: 'works with'  
    EndNodeID: 6
```

`relation` is a `Neo4jRelation` object with these properties:

- Relationship identifier
- Relationship data
- Start node identifier
- Relationship type
- End node identifier

Delete the relationship.

```
deleteRelation(neo4jconn, relation)
```

Delete the two nodes by using a `Neo4jNode` object array.

```
nodes = [startnode, endnode];  
deleteNode(neo4jconn, nodes)
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

`Neo4jConnect` object

Neo4j database connection, specified as a `Neo4jConnect` object created with the function `neo4j`.

relation — Relationship

`Neo4jRelation` object | `Neo4jRelation` object array | numeric scalar | numeric vector

Relationship in a Neo4j database, specified as a `Neo4jRelation` object, `Neo4jRelation` object array, numeric scalar, or numeric vector. For a single relationship, use a `Neo4jRelation` object or a numeric scalar that contains the relationship identifier. For multiple relationships, use a `Neo4jRelation` object array or a numeric vector that contains an array of relationship identifiers.

Example: 15

Example: [15, 16, 17]

Version History

Introduced in R2018a

See Also

neo4j | createRelation | createNode | updateRelation | close

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

updateNode

Package: database.neo4j.http

Update node labels and properties in Neo4j database

Syntax

```
updateNode(neo4jconn,node,'Labels',labels)
updateNode(neo4jconn,node,'Properties',properties)
updateNode(neo4jconn,node,'Labels',labels,'Properties',properties)
nodeinfo = updateNode( ___ )
```

Description

`updateNode(neo4jconn,node,'Labels',labels)` updates existing node labels with the specified node labels using a Neo4j database connection.

`updateNode(neo4jconn,node,'Properties',properties)` updates existing node properties with the specified node properties.

`updateNode(neo4jconn,node,'Labels',labels,'Properties',properties)` updates existing node labels and properties.

`nodeinfo = updateNode(___)` returns updated node information as a `Neo4jNode` object for one node, or as a table for multiple nodes, using any of the input argument combinations in the previous syntaxes.

Examples

Update Node Labels

Create a single node in a Neo4j® database and update its node labels.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Create a single node in the database using the Neo4j database connection.

```
node = createNode(neo4jconn)
```

```
node =
  Neo4jNode with properties:
      NodeID: 47
      NodeData: [1x1 struct]
      NodeLabels: []
```

node is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node label

Update the node by adding the labels `Person` and `Employee`.

```
labels = ["Person", "Employee"];
updateNode(neo4jconn, node, 'Labels', labels)
```

Display the updated node information. `nodeinfo` is a `Neo4jNode` object.

```
nodeid = node.NodeID;
nodeinfo = searchNodeByID(neo4jconn, nodeid);
nodeinfo.NodeLabels
```

```
ans = 2x1 cell array
    {'Person' }
    {'Employee'}
```

Close the database connection.

```
close(neo4jconn)
```

Update Properties of Existing Node

Search for an existing node in a Neo4j® database, add a node property, and display the updated node properties.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from `User1` through `User7`. Each relationship has the type `knows`.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url, username, password);
```

Check the `Message` property of the Neo4j connection object `neo4j conn`. The blank `Message` property indicates a successful connection.

```
neo4j conn.Message
```

```
ans =  
    []
```

Search for a node with the label `Person`. Then, using the Neo4j database connection, filter the results by the property key and value for the person named `User7`.

```
nlabel = 'Person';  
node = searchNode(neo4jconn,nlabel,'PropertyKey','name', ...  
    'PropertyValue','User7')
```

```
node =  
    Neo4jNode with properties:  
        NodeID: 9  
        NodeData: [1x1 struct]  
        NodeLabels: 'Person'
```

`node` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

Retrieve the existing properties of the node by using the `NodeData` property of the `Neo4jNode` object. `properties` is a structure.

```
properties = node.NodeData  
properties = struct with fields:  
    name: 'User7'
```

Update the properties of the node. Add another node property by setting a new field in the structure to specify the job title of the person.

```
properties.title = 'Analyst';  
updateNode(neo4jconn,node,'Properties',properties)
```

Display the updated node properties. `nodeinfo` is a `Neo4jNode` object.

```
nodeid = node.NodeID;  
nodeinfo = searchNodeByID(neo4jconn,nodeid);  
nodeinfo.NodeData  
ans = struct with fields:  
    name: 'User7'  
    title: 'Analyst'
```

Close the database connection.


```
close(neo4jconn)
```

Update Labels and Properties for One Node

Create a single node in a Neo4j® database, update its node labels and properties, and display them.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type knows.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Create a single node in the database using the Neo4j database connection.

```
node = createNode(neo4jconn)

node =
    Neo4jNode with properties:
        NodeID: 48
        NodeData: [1x1 struct]
        NodeLabels: []
```

`node` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

Create a table with one row that contains data about a person. Specify the name and job title of the person.

```
properties = table("User8","Analyst",'VariableNames',{'Name','Title'});
```

Update the node by adding the labels `Person` and `Employee` and the node properties defined in the table.

```
labels = ["Person", "Employee"];
updateNode(neo4jconn, node, 'Labels', labels, ...
           'Properties', properties)
```

Display the updated node labels. `nodeinfo` is a `Neo4jNode` object.

```
nodeid = node.NodeID;
nodeinfo = searchNodeByID(neo4jconn, nodeid);
nodeinfo.NodeLabels
```

```
ans = 2x1 cell array
    {'Person' }
    {'Employee'}
```

Display the updated node properties.

```
nodeinfo.NodeData

ans = struct with fields:
    Title: 'Analyst'
    Name: 'User8'
```

Close the database connection.

```
close(neo4jconn)
```

Update Labels and Properties for Two Nodes

Create two nodes in a Neo4j® database, update their node labels and properties, and display the labels and properties for the first node.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from `User1` through `User7`. Each relationship has the type `knows`.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url, username, password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =

    []
```

Create two nodes in the database using the Neo4j database connection. These nodes represent two people.

```
user8 = createNode(neo4jconn);
user9 = createNode(neo4jconn);
```

Create a table with two rows. Each row contains data about a person. Specify the name and job title for each person.

```
properties = table(["User8";"User9"],["Analyst";"Technician"], ...
    'VariableNames',{ 'Name', 'Title' });
```

Update the nodes by adding the labels Person and Employee and the node properties defined in the table.

```
labels = ["Person","Employee"];
updateNode(neo4jconn,[user8;user9], 'Labels', labels, ...
    'Properties', properties)
```

Display the node labels for the nodes.

```
nodeid = [user8.NodeID user9.NodeID];
nodeinfo = searchNodeByID(neo4jconn,nodeid);
nodeinfo.NodeLabels{:}
```

```
ans = 2x1 cell array
    {'Person' }
    {'Employee' }
```

```
ans = 2x1 cell array
    {'Person' }
    {'Employee' }
```

Display the node properties for the nodes.

```
nodeinfo.NodeData{:}
```

```
ans = struct with fields:
    Title: 'Analyst'
    Name: 'User8'
```

```
ans = struct with fields:
    Title: 'Technician'
    Name: 'User9'
```

Close the database connection.

```
close(neo4jconn)
```

Update Node Properties and Return Output

Create a single node in a Neo4j® database, update its properties, and display them. Access the updated node information using an output argument.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type knows.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Create a single node in the database using the Neo4j database connection.

```
node = createNode(neo4jconn)
```

```
node =
  Neo4jNode with properties:
      NodeID: 49
      NodeData: [1x1 struct]
      NodeLabels: []
```

`node` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

Update the properties of a node that represents a person. Create a table with one row that contains the name and job title for this person. The `nodeinfo` output argument is a `Neo4jNode` object.

```
properties = table("User8","Analyst",'VariableNames',{'Name','Title'});
nodeinfo = updateNode(neo4jconn,node,'Properties',properties);
```

Display the node properties.

```
nodeinfo.NodeData
ans = struct with fields:
    Title: 'Analyst'
    Name: 'User8'
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4j conn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

node — Node

Neo4jNode object | Neo4jNode object array | numeric scalar | numeric vector

Node in a Neo4j database, specified as a Neo4jNode object, Neo4jNode object array, numeric scalar, or a numeric vector. For one node, specify a Neo4jNode object or a numeric scalar. For multiple nodes, specify a Neo4jNode object array or a numeric vector.

The numeric scalar or vector must contain Neo4j database node identifiers.

Example: 15

Example: [2,3,4]

labels — Node labels

character vector | cell array of character vectors | string scalar | string array | cell array of cell arrays
| cell array of string arrays

Node labels, specified as a character vector, cell array of character vectors, string scalar, string array, cell array of cell arrays, or cell array of string arrays. To specify one node label, use a character vector or string scalar. For multiple node labels, use a cell array of character vectors or a string array. To update multiple nodes with different node labels, use a cell array of cell arrays or a cell array of string arrays.

Example: 'Person'

Data Types: char | string | cell

properties — Node properties

structure | structure array | table | cell array of structures

Node properties, specified as a structure, structure array, table, or cell array of structures.

When you specify a structure, the updateNode function converts each field and its corresponding value to a property and its corresponding value in the database node. The function also sets the NodeData property of the Neo4jNode object to this structure.

When you specify a table that contains one row, the updateNode function converts each variable and its corresponding value to a property and its corresponding value in the database node. The function also converts the variables and their corresponding values to fields and their corresponding values in a structure. The function sets this structure to the NodeData property of the Neo4jNode object.

To update multiple nodes, specify a structure array or table with multiple rows.

To update multiple nodes with different properties, specify a cell array of structures.

Note If a property is missing its corresponding value, then the updated node does not contain this property.

Data Types: `struct` | `table` | `cell`

Output Arguments

nodeinfo — Node information

Neo4jNode object | table

Node information in the Neo4j database, returned as a Neo4jNode object for one node or as a table for multiple nodes.

For multiple nodes, the table contains these variables:

- `NodeLabels` — Cell array of character vectors that contains the node labels for each database node
- `NodeData` — Cell array of structures that contains node information such as property keys
- `NodeObject` — Neo4jNode object for each database node

The row names of the table are Neo4j node identifiers of each database node.

Version History

Introduced in R2018a

See Also

`neo4j` | `createNode` | `searchNode` | `searchNodeByID` | `deleteNode` | `addNodeLabel` | `removeNodeLabel` | `removeNodeProperty` | `setNodeProperty` | `close`

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

updateRelation

Package: database.neo4j.http

Update relationship properties in Neo4j database

Syntax

```
updateRelation(neo4jconn,relation,properties)
relationinfo = updateRelation(neo4jconn,relation,properties)
```

Description

`updateRelation(neo4jconn,relation,properties)` updates the properties for one or more relationships in a Neo4j database using a Neo4j database connection.

`relationinfo = updateRelation(neo4jconn,relation,properties)` returns updated relationship information as a `Neo4jRelation` object for one relationship, or as a table for multiple relationships.

Examples

Update Relationship Between Two Nodes

Create a single relationship between two nodes in a Neo4j® database and update the properties of the relationship.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Create two nodes in the database using the Neo4j database connection. Use the `'Labels'` name-value pair argument to specify the label `Person` for each node.

```
label = 'Person';
startnode = createNode(neo4jconn,'Labels',label);
endnode = createNode(neo4jconn,'Labels',label);
```

Create a relationship between the two nodes using the Neo4j database connection. Specify the relationship type as `works with`. The output `relationinfo` is a `Neo4jRelation` object.

```
relationtype = 'works with';  
relationinfo = createRelation(neo4jconn,startnode,endnode,relationtype);
```

Update the relationship to include two more properties. The nodes represent two colleagues who started working together on a project named `Database` on September 1, 2017. Specify the project name and start date as properties of the relationship by using the `properties` structure.

```
properties.Project = 'Database';  
properties.StartDate = '09/01/2017';  
updateRelation(neo4jconn,relationinfo,properties)
```

Display the updated relationship information.

```
relationid = relationinfo.RelationID;  
relationinfo = searchRelationByID(neo4jconn,relationid);  
relationinfo.RelationType
```

```
ans =  
'works with'
```

```
relationinfo.RelationData
```

```
ans = struct with fields:  
  StartDate: '09/01/2017'  
  Project: 'Database'
```

Close the database connection.

```
close(neo4jconn)
```

Update Relationship Between Two Nodes and Return Output

Create a single relationship between two nodes in a Neo4j® database, update the properties of the relationship, and display the properties. Access the updated relationship information using an output argument.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';  
username = 'neo4j';  
password = 'matlab';  
  
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```



```
ans =
```

```
    []
```

Create two nodes in the database using the Neo4j database connection. Use the 'Labels' name-value pair argument to specify the label Person for each node.

```
label = 'Person';
startnode = createNode(neo4jconn, 'Labels', label);
endnode = createNode(neo4jconn, 'Labels', label);
```

Create a relationship between the two nodes using the Neo4j database connection. Specify the relationship type as works with. The output relationinfo is a Neo4jRelation object.

```
relationtype = 'works with';
relationinfo = createRelation(neo4jconn, startnode, endnode, relationtype);
```

Update the relationship to include two more properties. The nodes represent two colleagues who started working together on a project named Database on September 1, 2017. Specify the project name and start date as properties of the relationship by using the properties structure.

```
properties.Project = 'Database';
properties.StartDate = '09/01/2017';
relationinfo = updateRelation(neo4jconn, relationinfo, properties)
```

```
relationinfo =
  Neo4jRelation with properties:
```

```
    RelationID: 18
  RelationData: [1×1 struct]
   StartNodeID: 50
  RelationType: 'works with'
   EndNodeID: 51
```

relationinfo is a Neo4jRelation object that contains these properties:

- Relationship identifier
- Relationship data
- Start node identifier
- Relationship type
- End node identifier

Display the updated relationship properties.

```
relationinfo.RelationData
ans = struct with fields:
  StartDate: '09/01/2017'
  Project: 'Database'
```

Close the database connection.

```
close(neo4jconn)
```

Update Multiple Relationships

Create two relationships between nodes in a Neo4j® database, update the properties of the relationships, and display the properties.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type knows.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Search for the node with the label `Person` and the property key name set to the value `User7` by using the Neo4j database connection.

```
nlabel = 'Person';
user7 = searchNode(neo4jconn,nlabel,'PropertyKey','name', ...
    'PropertyValue','User7');
```

Create two nodes in the database using the Neo4j database connection. Use the `'Labels'` name-value pair argument to specify the label `Person` for each node.

```
label = 'Person';
user8 = createNode(neo4jconn,'Labels',label);
user9 = createNode(neo4jconn,'Labels',label);
```

Create two relationships using the Neo4j database connection. Specify the relationship types as `works with` and `studies with`. The two relationships are:

- User8 works with User7
- User8 studies with User9

`relationinfo` is a table that contains the relationship and node information.

```
startnode = [user8,user8];
endnode = [user7,user9];
relationtype = {'works with','studies with'};
relationinfo = createRelation(neo4jconn,startnode,endnode,relationtype);
```

Create a table that defines the properties. Here, User8 works with User7 in the workplace, and User8 studies with User9 in the library. Also, User8 started working with User7 on January 2, 2017, and User8 started studying with User9 on March 6, 2017.

```
properties = table(["Workplace";"Library"],["01/02/2017";"03/06/2017"], ...
  'VariableNames',{ 'Location', 'Date'});
```

Update both relationships with these properties.

```
relations = relationinfo.RelationObject;
relationinfo = updateRelation(neo4jconn,relations,properties)
```

```
relationinfo=2x5 table
      StartNodeID      RelationType      EndNodeID      RelationData      RelationObject
-----
      12          17      'works with'          9      [1x1 struct]      [1x1 database.neo4j.http
      11          17      'studies with'       18     [1x1 struct]      [1x1 database.neo4j.http
```

relationinfo is a table with these variables:

- Start node identifier
- Relationship type
- End node identifier
- Relationship data
- Neo4jRelation object

Display the updated properties of the relationships.

```
relationinfo.RelationData{1}
```

```
ans = struct with fields:
      Date: '01/02/2017'
      Location: 'Workplace'
```

```
relationinfo.RelationData{2}
```

```
ans = struct with fields:
      Date: '03/06/2017'
      Location: 'Library'
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

relation — Relationship

Neo4jRelation object | Neo4jRelation object array | numeric scalar | numeric vector

Relationship in a Neo4j database, specified as a `Neo4jRelation` object, `Neo4jRelation` object array, numeric scalar, or numeric vector. For a single relationship, use a `Neo4jRelation` object or a numeric scalar that contains the relationship identifier. For multiple relationships, use a `Neo4jRelation` object array or a numeric vector that contains an array of relationship identifiers.

Example: 15

Example: [15,16,17]

properties — Relationship properties

structure | structure array | table | cell array of structures

Relationship properties, specified as a structure, structure array, table, or cell array of structures.

When you specify a structure, the `updateRelation` function converts each field and its corresponding value to a property and its corresponding value in the database relationship. When you specify a table with one row, the function converts each variable and its corresponding value to a property and its corresponding value in the database relationship.

The `updateRelation` function also sets the `RelationObject` variable of the `relationinfo` output argument to the `Neo4jRelation` object, which contains the relationship information.

For multiple relationships, specify a structure array or table with multiple rows.

For multiple relationships with different properties, specify a cell array of structures.

Note If a property is missing its corresponding value, then the updated relationship does not contain this property.

Data Types: struct | table | cell

Output Arguments**relationinfo — Relationship information**

Neo4jRelation object | table

Relationship information, returned as a `Neo4jRelation` object for one relationship or as a table for multiple relationships.

For multiple relationships, the table contains these variables:

- `StartNodeID` — Node identifier of the start node for each matched relationship
- `RelationType` — Character vector that denotes the relationship type for each matched relationship
- `EndNodeID` — Node identifier of the end node for each matched relationship
- `RelationData` — Structure array that contains property keys associated with each matched relationship
- `RelationObject` — `Neo4jRelation` object for each matched relationship

The row names in the table are Neo4j relationship identifiers.

Version History

Introduced in R2018a

See Also

neo4j | createRelation | deleteRelation | setRelationProperty |
removeRelationProperty | close | createNode | searchNode | searchRelationByID

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

searchRelationByID

Package: database.neo4j.http

Search Neo4j relationship by relationship identifier

Syntax

```
relationinfo = searchRelationByID(neo4jconn,relationid)
```

Description

`relationinfo = searchRelationByID(neo4jconn,relationid)` returns the Neo4j relationship specified by the relationship identifier using the Neo4j database connection.

Examples

Search for Relationships

Search for a single relationship or multiple relationships by using relationship identifiers in the Neo4j® database.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from User1 through User7. Each relationship has the type knows.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Search for the relationship with the identifier 8 by using the Neo4j database connection.

```
relationid = 8;
relationinfo = searchRelationByID(neo4jconn,relationid)
```

```
relationinfo =
  Neo4jRelation with properties:
```

```

RelationID: 8
RelationData: [1x1 struct]
StartNodeID: 5
RelationType: 'knows'
EndNodeID: 9

```

relationinfo is a Neo4jRelation object with these properties:

- Relationship identifier
- Relationship data
- Start node identifier
- Relationship type
- End node identifier

Display the relationship type.

```
relationinfo.RelationType
```

```
ans =
'knows'
```

Search for multiple relationships with the identifiers 4, 5, and 6 by using the Neo4j database connection.

```
relationid = [4,5,6];
relationinfo = searchRelationByID(neo4jconn, relationid)
```

```
relationinfo=3x5 table
```

	StartNodeID	RelationType	EndNodeID	RelationData	RelationObject
5	3	'knows'	4	[1x1 struct]	[1x1 database.neo4j.http.Neo4jRelation]
4	3	'knows'	5	[1x1 struct]	[1x1 database.neo4j.http.Neo4jRelation]
6	5	'knows'	4	[1x1 struct]	[1x1 database.neo4j.http.Neo4jRelation]

relationinfo is a table with these variables:

- Start node identifier
- Relationship type
- End node identifier
- Relationship data
- Neo4jRelation object

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a `Neo4jConnect` object created with the function `neo4j`.

relationid — Relationship identifier

numeric scalar | numeric vector

Relationship identifier, specified as a numeric scalar for a single relationship or numeric vector for multiple relationships.

Example: `[15, 16]`

Data Types: `double`

Output Arguments

relationinfo — Relationship information

`Neo4jRelation` object | table

Relationship information, returned as a `Neo4jRelation` object for one relationship or as a table for multiple relationships.

For multiple relationships, the table contains these variables:

- `StartNodeID` — Node identifier of the start node for each matched relationship
- `RelationType` — Character vector that denotes the relationship type for each matched relationship
- `EndNodeID` — Node identifier of the end node for each matched relationship
- `RelationData` — Structure array that contains property keys associated with each matched relationship
- `RelationObject` — `Neo4jRelation` object for each matched relationship

The row names in the table are Neo4j relationship identifiers.

Version History

Introduced in R2018a

See Also

`neo4j` | `searchNodeByID` | `searchRelation` | `close`

Topics

“Explore Graph Database Structure” on page 9-2

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

storeDigraph

Package: database.neo4j.http

Store directed graph in Neo4j database

Syntax

```
storeDigraph(neo4jconn,G)
storeDigraph(neo4jconn,G,Name,Value)
graphinfo = storeDigraph( ___ )
```

Description

`storeDigraph(neo4jconn,G)` converts a directed graph to a Neo4j graph and stores it in a Neo4j database using a Neo4j database connection. The variables in the node and edge tables of the `digraph` object (except the `EndNodes` variable) become the properties of the nodes and relationships in the Neo4j graph.

`storeDigraph(neo4jconn,G,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'GlobalNodeLabel', 'Person'` stores all nodes in the directed graph by using the `Person` node label.

`graphinfo = storeDigraph(___)` returns graph information as a structure using any of the input argument combinations in the previous syntaxes.

Examples

Store Directed Graph in Neo4j Database

Create a `digraph` object and store its contents in a Neo4j® database. Display the contents of the resulting Neo4j graph.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from `User1` through `User7`. Each relationship has the type `knows`.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
```

```
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
    []
```

Create a `digraph` object with three nodes, which represents a new Neo4j graph. The nodes represent three additional people: `User8`, `User9`, and `User10`.

```
G = digraph([1 1 3],[2 3 2],[1 2 3],{'User8','User9','User10'});
```

Store the data as a Neo4j graph in the Neo4j database.

```
storeDigraph(neo4jconn,G)
```

By default, the `storeDigraph` function stores the directed graph without node labels. Also, the function stores the relationships with the default relationship type `Edge`.

Display information about the Neo4j graph nodes. `graphinfo` is a structure that contains node and relationship information.

```
criteria = ["Edge"];
graphinfo = searchGraph(neo4jconn,criteria);
graphinfo.Nodes
```

```
ans=3x3 table
      NodeLabels      NodeData      NodeObject
      _____      _____      _____
      7             []             [1x1 struct]   [1x1 database.neo4j.http.Neo4jNode]
      52            []             [1x1 struct]   [1x1 database.neo4j.http.Neo4jNode]
      47            []             [1x1 struct]   [1x1 database.neo4j.http.Neo4jNode]
```

`Nodes` is a table that contains these variables:

- Node label
- Node data
- `Neo4jNode` object

Display information about the Neo4j graph relationships.

```
graphinfo.Relations
```

```
ans=3x5 table
      StartNodeID      RelationType      EndNodeID      RelationData      RelationObject
      _____      _____      _____      _____      _____
      17             7             'Edge'         52             [1x1 struct]   [1x1 database.neo4j.http.Neo4jNode]
      18             47            'Edge'         52             [1x1 struct]   [1x1 database.neo4j.http.Neo4jNode]
      35             7             'Edge'         47             [1x1 struct]   [1x1 database.neo4j.http.Neo4jNode]
```

`Relations` is a table that contains these variables:

- Start node identifier
- Relationship type
- End node identifier

- Relationship data
- Neo4jRelation object

Close the database connection.

```
close(neo4jconn)
```

Store Directed Graph with Global Node Labels and Relationship Types

Create a `digraph` object and store its contents in a Neo4j® database. Specify a node label to apply to all nodes in the resulting Neo4j graph. Specify a relationship type to apply to all relationships in the resulting Neo4j graph. Display the contents of the graph.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from `User1` through `User7`. Each relationship has the type `knows`.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Create a `digraph` object with three nodes, which represents a new Neo4j graph. The nodes represent three additional people: `User8`, `User9`, and `User10`.

```
G = digraph([1 1 3],[2 3 2],[1 2 3],["User8" "User9" "User10"]);
```

Store the data as a Neo4j graph in the Neo4j database. Specify the node label `Person` for each node in the resulting Neo4j graph by using the `'GlobalNodeLabel'` name-value pair argument. Specify the relationship type `knows` for each relationship in the graph by using the `'GlobalRelationshipType'` name-value pair argument.

```
storeDigraph(neo4jconn,G,'GlobalNodeLabel','Person', ...
    'GlobalRelationshipType','knows')
```

Display information about the Neo4j graph nodes. `graphinfo` is a structure that contains node and relationship information.

```
criteria = {'Person'};
graphinfo = searchGraph(neo4jconn,criteria);
graphinfo.Nodes
```

```
ans=10x3 table
```

	NodeLabels	NodeData	NodeObject
0	'Person'	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
48	'Person'	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
1	'Person'	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
2	'Person'	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
3	'Person'	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
4	'Person'	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
5	'Person'	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
9	'Person'	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
49	'Person'	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
50	'Person'	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]

Nodes is a table that contains these variables:

- Node label
- Node data
- Neo4jNode object

graphinfo contains the three additional nodes.

Display information about the Neo4j graph relationships.

graphinfo.Relations

```
ans=11x5 table
```

	StartNodeID	RelationType	EndNodeID	RelationData	RelationObject
1	0	'knows'	1	[1x1 struct]	[1x1 database.neo4j.http.Ne
0	0	'knows'	2	[1x1 struct]	[1x1 database.neo4j.http.Ne
3	1	'knows'	3	[1x1 struct]	[1x1 database.neo4j.http.Ne
2	2	'knows'	1	[1x1 struct]	[1x1 database.neo4j.http.Ne
5	3	'knows'	4	[1x1 struct]	[1x1 database.neo4j.http.Ne
4	3	'knows'	5	[1x1 struct]	[1x1 database.neo4j.http.Ne
6	5	'knows'	4	[1x1 struct]	[1x1 database.neo4j.http.Ne
8	5	'knows'	9	[1x1 struct]	[1x1 database.neo4j.http.Ne
19	48	'knows'	49	[1x1 struct]	[1x1 database.neo4j.http.Ne
7	48	'knows'	50	[1x1 struct]	[1x1 database.neo4j.http.Ne
9	50	'knows'	49	[1x1 struct]	[1x1 database.neo4j.http.Ne

Relations is a table that contains these variables:

- Start node identifier
- Relationship type
- End node identifier
- Relationship data
- Neo4jRelation object

graphinfo contains the three additional relationships.

Close the database connection.

```
close(neo4jconn)
```

Store Directed Graph with Node Labels and Relationship Types and Return Output

Create a `digraph` object by specifying nodes and edges. Then, store the directed graph in a Neo4j® database by specifying node labels and relationship types. Display the contents of the resulting Neo4j graph. Access the graph information using an output argument.

Assume that you have graph data stored in a Neo4j database that represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key name with a value ranging from `User1` through `User7`. Each relationship has the type `knows`.

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j® connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Create a table for nodes. Define the `names` variable, which contains the names of three additional people: `User8`, `User9`, and `User10`. Then, define the `classification` variable to classify each person as `Person`. Also, define the `titles` variable, which contains the job title of each person. The first two people are analysts and the third is a technician.

```
names = {'User8';'User9';'User10'};
classification = {'Person';'Person';'Person'};
titles = {'Analyst';'Analyst';'Technician'};
nodetable = table(names,classification,titles,'VariableNames', ...
    {'Name','Classification','Title'});
```

Create a table with two edges. One edge specifies that two people know each other. The other edge specifies that two people work with each other.

```
edge1 = [1 2];
edge2 = [3 3];
description = {'knows','works with'};
edgetable = table([edge1',edge2'],description', ...
    'VariableNames',{'EndNodes','Description'});
```

Create a `digraph` object using the edge and node tables.

```
G = digraph(edgetable,nodetable);
```

Store the data in the `digraph` object as a Neo4j graph in the Neo4j database. Specify the node labels for each node in the resulting Neo4j graph by using the `'NodeLabel'` name-value pair argument.

The graph uses the `Classification` and `Title` variables of the node table for the node labels. Also, the graph uses the `Description` variable of the edge table for the relationship types.

```
labels = {'Classification';'Title'};
relation = 'Description';
graphinfo = storeDigraph(neo4jconn,G,'NodeLabel',labels, ...
    'RelationType',relation)

graphinfo = struct with fields:
    Nodes: [3x3 table]
    Relations: [2x5 table]
```

Display information about the Neo4j graph nodes.

```
graphinfo.Nodes
```

```
ans=3x3 table
```

	NodeLabels	NodeData	NodeObject
6	{2x1 cell}	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
52	{2x1 cell}	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]
7	{2x1 cell}	[1x1 struct]	[1x1 database.neo4j.http.Neo4jNode]

Nodes is a table that contains these variables:

- Node label
- Node data
- Neo4jNode object

Display information about the Neo4j graph relationships.

```
graphinfo.Relations
```

```
ans=2x5 table
```

	StartNodeID	RelationType	EndNodeID	RelationData	RelationObject
17	6	'knows'	7	[1x1 struct]	[1x1 database.neo4j.http.Ne
35	52	'works with'	7	[1x1 struct]	[1x1 database.neo4j.http.Ne

Relations is a table that contains these variables:

- Start node identifier
- Relationship type
- End node identifier
- Relationship data
- Neo4jRelation object

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

G — Directed graph

digraph object

Directed graph, specified as a digraph object.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: graphinfo = storeDigraph(neo4jconn,G, 'GlobalNodeLabel', 'Person', 'GlobalRelationType', 'knows') stores a directed graph and specifies that all nodes in the resulting graph have the Person node label and all relationships have the knows type.

Note If you do not specify 'GlobalNodeLabel' or 'NodeLabel', the resulting Neo4j graph contains nodes without labels.

GlobalNodeLabel — Global node label

character vector | cell array of character vectors | string scalar | string array

Global node label, specified as the comma-separated pair consisting of 'GlobalNodeLabel' and a character vector, cell array of character vectors, string scalar, or string array. To specify one node label, use a character vector or string scalar. To specify multiple node labels, use a cell array of character vectors or a string array.

After you execute the storeDigraph function, each node in the resulting Neo4j graph contains node labels that you specify using this name-value pair argument.

Example: "Person"

Example: {'Person', 'Employee'}

Data Types: char | string | cell

NodeLabel — Node label

character vector | cell array of character vectors | string scalar | string array

Node label, specified as the comma-separated pair consisting of 'NodeLabel' and a character vector, cell array of character vectors, string scalar, or string array. To specify one node label, use a character vector or string scalar. To specify multiple node labels, specify a cell array of character vectors or a string array.

To specify different labels for nodes in the resulting Neo4j graph, use this name-value pair argument. The specified node labels must match the variable names in the table of node information in the `digraph` object.

Example: "Person"

Example: {'Name', 'Title'}

Data Types: char | string | cell

GlobalRelationshipType – Global relationship type

'Edge' (default) | character vector | string scalar

Global relationship type, specified as the comma-separated pair consisting of 'GlobalRelationshipType' and a character vector or string scalar. To specify the same type of relationship for all relationships between nodes in the resulting Neo4j graph, use this name-value pair argument.

Note When specifying the type of relationship, use either the 'GlobalRelationshipType' or 'RelationshipType' name-value pair argument. You cannot specify both of these arguments at the same time.

Example: "knows"

Data Types: char | string

RelationshipType – Relationship type

'Edge' (default) | character vector | string scalar

Relationship type, specified as the comma-separated pair consisting of 'RelationshipType' and a character vector or string scalar. To specify different types of relationships between nodes in the resulting Neo4j graph, use this name-value pair argument. The specified types must match the variable names in the table of edge information in the `digraph` object.

Note When specifying the type of relationship, use either the 'RelationshipType' or 'GlobalRelationshipType' name-value pair argument. You cannot specify both of these arguments at the same time.

Example: 'Description'

Data Types: char | string

Output Arguments

graphinfo – Graph information

structure

Graph information in the Neo4j database, returned as a structure with these fields.

Field	Description
Nodes	<p>Table that contains node information for each node in the Relations table. The Nodes table contains the following fields:</p> <ul style="list-style-type: none"> • <code>NodeLabels</code> — Character vector that denotes the node label • <code>NodeData</code> — Structure array that contains node information such as property keys • <code>NodeObject</code> — <code>Neo4jNode</code> object that represents each node <p>The row names in the table are Neo4j node identifiers.</p>
Relations	<p>Table that contains relationship information for the nodes in the Nodes table. The Relations table contains the following fields:</p> <ul style="list-style-type: none"> • <code>StartNodeID</code> — Node identifier for the start node of each relationship • <code>RelationType</code> — Character vector that denotes the type of each relationship • <code>EndNodeID</code> — Node identifier for the end node of each relationship • <code>RelationData</code> — Structure array that contains property keys associated with each relationship • <code>RelationObject</code> — <code>Neo4jRelation</code> object that represents each relationship <p>The row names in the table are Neo4j relationship identifiers.</p>

Tips

- The `storeDigraph` function stores all MATLAB objects as JSON string equivalents in the Neo4j graph. For example, the function stores the date `datetime('Jan/01/2017')` as "Jan/01/2017" in the Neo4j graph.

Version History

Introduced in R2018a

See Also

`deleteNode` | `neo4j` | `deleteRelation` | `createNode` | `createRelation` | `addNodeLabel` | `removeNodeLabel` | `removeNodeProperty` | `removeRelationProperty` | `setNodeProperty` | `setRelationProperty` | `close`

Topics

"Add and Query Group of Colleagues in Social Neighborhood" on page 9-14
 "Graph Database Workflow for Neo4j Database Interfaces" on page 9-6

addNodeLabel

Package: database.neo4j.http

Add labels to nodes in Neo4j database

Syntax

```
addNodeLabel(neo4jconn,node,labels)
nodeinfo = addNodeLabel(neo4jconn,node,labels)
```

Description

`addNodeLabel(neo4jconn,node,labels)` adds node labels to one or more nodes in a Neo4j database using a Neo4j database connection.

`nodeinfo = addNodeLabel(neo4jconn,node,labels)` returns updated node information as a Neo4jNode object for one node, or as a table for multiple nodes.

Examples

Add One Node Label

Add one node label to a single node in a Neo4j® database and access the node.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Retrieve the first node in the table of node information. This node has the label `Person`.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn,nlabel);
node = nodeinfo.NodeObject(1);
```

Add one node label to a single node in the database using the Neo4j database connection.

```
labels = "Analyst";
addNodeLabel(neo4jconn,node,labels)
```

Display the node labels for the updated node. The `NodeLabels` property contains two labels.

```
nodeinfo = searchNode(neo4jconn, labels);
nodeinfo.NodeLabels

ans = 2x1 cell array
    {'Person' }
    {'Analyst' }
```

Close the database connection.

```
close(neo4jconn)
```

Add Multiple Node Labels and Return Output

Add node labels to multiple nodes in a Neo4j® database. Access the updated node information using an output argument.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url, username, password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message

ans =

    []
```

Find nodes with the label `Person`, and display the node labels.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn, nlabel);
nodeinfo.NodeLabels

ans = 7x1 cell array
    {'Person'}
    {'Person'}
    {'Person'}
    {'Person'}
    {'Person'}
    {'Person'}
    {'Person'}
```

Add multiple node labels to the nodes in the database using the Neo4j database connection. The `nodeinfo` output argument is a `Neo4jNode` object.

```
node = nodeinfo.NodeObject;  
labels = ["Analyst" "Scientist"];  
nodeinfo = addNodeLabel(neo4jconn,node,labels);
```

Display the node information for the updated nodes. Each node has three node labels (Person, Analyst, and Scientist).

```
nodeinfo.NodeLabels
```

```
ans = 7×1 cell array  
    {3×1 cell}  
    {3×1 cell}  
    {3×1 cell}  
    {3×1 cell}  
    {3×1 cell}  
    {3×1 cell}  
    {3×1 cell}
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

node — Node

Neo4jNode object | Neo4jNode object array | numeric scalar | numeric vector

Node in a Neo4j database, specified as a Neo4jNode object, Neo4jNode object array, numeric scalar, or a numeric vector. For one node, specify a Neo4jNode object or a numeric scalar. For multiple nodes, specify a Neo4jNode object array or a numeric vector.

The numeric scalar or vector must contain Neo4j database node identifiers.

Example: 15

Example: [2,3,4]

labels — Node labels

character vector | cell array of character vectors | string scalar | string array

Node labels, specified as a character vector, cell array of character vectors, string scalar, or string array. To specify one node label, use a character vector or string scalar. For multiple node labels, use a cell array of character vectors or a string array.

Example: "Person"

Data Types: char | string | cell

Output Arguments

nodeinfo — Node information

Neo4jNode object | table

Node information in the Neo4j database, returned as a Neo4jNode object for one node or as a table for multiple nodes.

For multiple nodes, the table contains these variables:

- **NodeLabels** — Cell array of character vectors that contains the node labels for each database node
- **NodeData** — Cell array of structures that contains node information such as property keys
- **NodeObject** — Neo4jNode object for each database node

The row names of the table are Neo4j node identifiers of each database node.

Version History

Introduced in R2019a

See Also

neo4j | searchNode | removeNodeLabel | setNodeProperty | removeNodeProperty | close

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

removeNodeLabel

Package: database.neo4j.http

Remove labels from nodes in Neo4j database

Syntax

```
removeNodeLabel(neo4jconn,node,labels)
nodeinfo = removeNodeLabel(neo4jconn,node,labels)
```

Description

`removeNodeLabel(neo4jconn,node,labels)` removes node labels from one or more nodes in a Neo4j database using a Neo4j database connection.

`nodeinfo = removeNodeLabel(neo4jconn,node,labels)` returns updated node information as a `Neo4jNode` object for one node, or as a table for multiple nodes.

Examples

Remove One Node Label

Add one node label to a single node in a Neo4j® database, access the node, and then remove the new node label.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Retrieve the first node in the table of node information. This node has the label `Person`.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn,nlabel);
node = nodeinfo.NodeObject(1);
```

Add one node label to a single node in the database using the Neo4j database connection.

```
labels = "Analyst";
addNodeLabel(neo4jconn,node,labels)
```

Display the node information for the updated node.

```
nodeinfo = searchNode(neo4jconn,labels);
node.NodeLabels
```

```
ans = 2x1 cell array
    {'Person' }
    {'Analyst' }
```

Remove the label Analyst from the updated node.

```
removeNodeLabel(neo4jconn,node,labels)
```

Find the first node again and display its node labels. The node now has only the label Person.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn,nlabel);
nodeinfo.NodeLabels(1)
```

```
ans = 1x1 cell array
    {'Person' }
```

Close the database connection.

```
close(neo4jconn)
```

Remove Multiple Node Labels and Return Output

Add node labels to multiple nodes in a Neo4j® database, remove the new node labels, and access the updated node information using an output argument.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the Message property of the Neo4j connection object `neo4jconn`. The blank Message property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Find nodes with the label Person.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn,nlabel);
```

Add multiple node labels to the nodes in the database using the Neo4j database connection.

```
labels = ["Analyst" "Scientist"];
node = nodeinfo.NodeObject;
addNodeLabel(neo4jconn,node,labels)
```

Display the node information for the updated nodes. Each node has three node labels (Person, Analyst, and Scientist).

```
nlabel = "Analyst";
nodeinfo = searchNode(neo4jconn,nlabel);
nodeinfo.NodeLabels
```

```
ans = 7×1 cell array
    {3×1 cell}
    {3×1 cell}
    {3×1 cell}
    {3×1 cell}
    {3×1 cell}
    {3×1 cell}
    {3×1 cell}
```

Remove the labels Analyst and Scientist from the updated nodes and display updated node information. Each node now has only the label Person. The nodeinfo output argument is a Neo4jNode object.

```
nodeinfo = removeNodeLabel(neo4jconn,node,labels);
nodeinfo.NodeLabels
```

```
ans = 7×1 cell array
    {'Person'}
    {'Person'}
    {'Person'}
    {'Person'}
    {'Person'}
    {'Person'}
    {'Person'}
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

node — Node

Neo4jNode object | Neo4jNode object array | numeric scalar | numeric vector

Node in a Neo4j database, specified as a Neo4jNode object, Neo4jNode object array, numeric scalar, or a numeric vector. For one node, specify a Neo4jNode object or a numeric scalar. For multiple nodes, specify a Neo4jNode object array or a numeric vector.

The numeric scalar or vector must contain Neo4j database node identifiers.

Example: 15

Example: [2,3,4]

Labels — Node labels

character vector | cell array of character vectors | string scalar | string array

Node labels, specified as a character vector, cell array of character vectors, string scalar, or string array. To specify one node label, use a character vector or string scalar. For multiple node labels, use a cell array of character vectors or a string array.

Example: "Person"

Data Types: char | string | cell

Output Arguments

nodeinfo — Node information

Neo4jNode object | table

Node information in the Neo4j database, returned as a Neo4jNode object for one node or as a table for multiple nodes.

For multiple nodes, the table contains these variables:

- **NodeLabels** — Cell array of character vectors that contains the node labels for each database node
- **NodeData** — Cell array of structures that contains node information such as property keys
- **NodeObject** — Neo4jNode object for each database node

The row names of the table are Neo4j node identifiers of each database node.

Version History

Introduced in R2019a

See Also

neo4j | searchNode | addNodeLabel | setNodeProperty | removeNodeProperty | close

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

setNodeProperty

Package: database.neo4j.http

Set properties for nodes in Neo4j database

Syntax

```
setNodeProperty(neo4jconn,node,properties)
nodeinfo = setNodeProperty(neo4jconn,node,properties)
```

Description

`setNodeProperty(neo4jconn,node,properties)` sets properties for one or more nodes in a Neo4j database using a Neo4j database connection.

`nodeinfo = setNodeProperty(neo4jconn,node,properties)` returns updated node information as a `Neo4jNode` object for one node, or as a table for multiple nodes.

Examples

Set One Node Property

Set one node property for a single node in a Neo4j® database and access the node.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Retrieve the first node in the table of node information. This node has the label `Person`.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn,nlabel);
node = nodeinfo.NodeObject(1);
```

Set the `Title` node property for a single node in the database using the Neo4j database connection.

```
properties.Title = "Analyst";
setNodeProperty(neo4jconn,node,properties)
```

Display the node information for the updated node.

```
nodeinfo = searchNode(neo4jconn,nlabel);
node = nodeinfo.NodeObject(1);
node.NodeData

ans = struct with fields:
    name: 'User1'
    Title: 'Analyst'
```

Close the database connection.

```
close(neo4jconn)
```

Set Multiple Node Properties and Return Output

Set node properties for multiple nodes in a Neo4j® database. Access the updated node information using an output argument.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =

    []
```

Find nodes with the label `Person`, and select the first two nodes.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn,nlabel);
nodes = nodeinfo.NodeObject;
firstnodes = nodes(1:2);
```

Set the `Title` node property for multiple nodes to different values using the Neo4j database connection. Create a structure array to store the properties. Display the updated node information for the first two nodes. The `nodeinfo` output argument is a `Neo4jNode` object.

```
properties(1).Title = "Analyst";
properties(2).Title = "Engineer";
nodeinfo = setNodeProperty(neo4jconn,firstnodes,properties);
nodeinfo.NodeData{1:2}

ans = struct with fields:
    name: 'User1'
```

```
Title: 'Analyst'

ans = struct with fields:
  name: 'User3'
  Title: 'Engineer'
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

node — Node

Neo4jNode object | Neo4jNode object array | numeric scalar | numeric vector

Node in a Neo4j database, specified as a Neo4jNode object, Neo4jNode object array, numeric scalar, or a numeric vector. For one node, specify a Neo4jNode object or a numeric scalar. For multiple nodes, specify a Neo4jNode object array or a numeric vector.

The numeric scalar or vector must contain Neo4j database node identifiers.

Example: 15

Example: [2,3,4]

properties — Node properties

structure | structure array | table

Node properties, specified as a structure, structure array, or table.

If a property does not exist, then the `setNodeProperty` function adds a new property. If the property exists, then the function sets a new value for the existing property.

When you specify a structure, the `setNodeProperty` function converts each field and its corresponding value to a property and its corresponding value in the database node. When you specify a table that contains one row, the function converts each variable and its corresponding value to a property and its corresponding value in the database node.

Specify a structure array or a table with multiple rows to update multiple nodes in the database.

The dimensions of the data in the structure array or table must be the same as the number of the specified nodes in the database to update. However, you can use a scalar structure to set the same values for multiple nodes in the database simultaneously.

Data Types: struct | table

Output Arguments

nodeinfo — Node information

Neo4jNode object | table

Node information in the Neo4j database, returned as a Neo4jNode object for one node or as a table for multiple nodes.

For multiple nodes, the table contains these variables:

- **NodeLabels** — Cell array of character vectors that contains the node labels for each database node
- **NodeData** — Cell array of structures that contains node information such as property keys
- **NodeObject** — Neo4jNode object for each database node

The row names of the table are Neo4j node identifiers of each database node.

Version History

Introduced in R2019a

See Also

neo4j | searchNode | removeNodeProperty | addNodeLabel | removeNodeLabel | close

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

removeNodeProperty

Package: database.neo4j.http

Remove properties from nodes in Neo4j database

Syntax

```
removeNodeProperty(neo4jconn, node, propertyNames)
nodeinfo = removeNodeProperty(neo4jconn, node, propertyNames)
```

Description

`removeNodeProperty(neo4jconn, node, propertyNames)` removes properties from one or more nodes in a Neo4j database using a Neo4j database connection.

`nodeinfo = removeNodeProperty(neo4jconn, node, propertyNames)` returns updated node information as a `Neo4jNode` object for one node, or as a table for multiple nodes.

Examples

Remove One Node Property

Remove one node property from a single node in a Neo4j® database and access the node.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url, username, password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Retrieve the first node in the table of node information. This node has the label `Person`.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn, nlabel);
node = nodeinfo.NodeObject(1);
```

Set the `Title` node property for a single node in the database using the Neo4j database connection.

```
properties.Title = "Analyst";
setNodeProperty(neo4jconn, node, properties)
```

Display the node information for the updated node.

```
nodeinfo = searchNode(neo4jconn,nlabel);
node = nodeinfo.NodeObject(1);
node.NodeData

ans = struct with fields:
    name: 'User1'
    Title: 'Analyst'
```

Remove the node property.

```
propertyNames = "Title";
removeNodeProperty(neo4jconn,node,propertyNames)
```

Display the node information for the updated node.

```
nodeinfo = searchNode(neo4jconn,nlabel);
node = nodeinfo.NodeObject(1);
node.NodeData

ans = struct with fields:
    name: 'User1'
```

Close the database connection.

```
close(neo4jconn)
```

Remove Multiple Node Properties and Return Output

Remove node properties from multiple nodes in a Neo4j® database. Access the updated node information using an output argument.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =

    []
```

Find nodes with the label `Person`.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn,nlabel);
nodes = nodeinfo.NodeObject;
```

Set the `Title` and `Department` node properties for multiple nodes using the Neo4j database connection. Display the updated node information for the first three nodes.

```
properties.Title = "Analyst";
properties.Department = "Sales";
nodeinfo = setNodeProperty(neo4jconn,nodes,properties);
nodeinfo.NodeData{1:3}
```

```
ans = struct with fields:
    Department: 'Sales'
    name: 'User1'
    Title: 'Analyst'
```

```
ans = struct with fields:
    Department: 'Sales'
    name: 'User3'
    Title: 'Analyst'
```

```
ans = struct with fields:
    Department: 'Sales'
    name: 'User2'
    Title: 'Analyst'
```

Remove the node properties using the property names. Display the updated node information for the first three nodes. The `nodeinfo` output argument is a `Neo4jNode` object.

```
propertyNames = ["Title" "Department"];
nodeinfo = removeNodeProperty(neo4jconn,nodes,propertyNames);
nodeinfo.NodeData{1:3}
```

```
ans = struct with fields:
    name: 'User1'
```

```
ans = struct with fields:
    name: 'User3'
```

```
ans = struct with fields:
    name: 'User2'
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a `Neo4jConnect` object created with the function `neo4j`.

node — Node

Neo4jNode object | Neo4jNode object array | numeric scalar | numeric vector

Node in a Neo4j database, specified as a `Neo4jNode` object, `Neo4jNode` object array, numeric scalar, or a numeric vector. For one node, specify a `Neo4jNode` object or a numeric scalar. For multiple nodes, specify a `Neo4jNode` object array or a numeric vector.

The numeric scalar or vector must contain Neo4j database node identifiers.

Example: 15

Example: [2,3,4]

propertyNames — Property names

character vector | cell array of character vectors | string scalar | string array

Property names, specified as a character vector, cell array of character vectors, string scalar, or string array. For one property, use a character vector or string scalar. For multiple properties, use a cell array of character vectors or a string array.

Example: "Analyst"

Example: ["Analyst" "Clerk"]

Data Types: char | string

Output Arguments

nodeinfo — Node information

`Neo4jNode` object | table

Node information in the Neo4j database, returned as a `Neo4jNode` object for one node or as a table for multiple nodes.

For multiple nodes, the table contains these variables:

- `NodeLabels` — Cell array of character vectors that contains the node labels for each database node
- `NodeData` — Cell array of structures that contains node information such as property keys
- `NodeObject` — `Neo4jNode` object for each database node

The row names of the table are Neo4j node identifiers of each database node.

Version History

Introduced in R2019a

See Also

`neo4j` | `searchNode` | `setNodeProperty` | `addNodeLabel` | `removeNodeLabel` | `close`

Topics

"Update Friend Information in Social Neighborhood" on page 9-11

"Graph Database Workflow for Neo4j Database Interfaces" on page 9-6

setRelationProperty

Package: database.neo4j.http

Set properties for relationships in Neo4j database

Syntax

```
setRelationProperty(neo4jconn, relation, properties)
relationinfo = setRelationProperty(neo4jconn, relation, properties)
```

Description

`setRelationProperty(neo4jconn, relation, properties)` sets properties for one or more relationships in a Neo4j database using a Neo4j database connection.

`relationinfo = setRelationProperty(neo4jconn, relation, properties)` returns updated relationship information as a `Neo4jRelation` object for one relationship, or as a table for multiple relationships.

Examples

Set One Relationship Property

Set one property for a single relationship in a Neo4j® database and access the relationship.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url, username, password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

Retrieve the second node in the table of node information. This node has the label `Person`.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn, nlabel);
node = nodeinfo.NodeObject(2);
```

Find the outgoing relationship from the origin node.

```
direction = "out";
relinfo = searchRelation(neo4jconn,node,direction);
relation = relinfo.Relations.RelationObject;
```

Set the StartDate property for a single relationship in the database using the Neo4j database connection.

```
properties.StartDate = "01/01/2018";
setRelationProperty(neo4jconn,relation,properties)
```

Display the relationship information for the updated relationship.

```
relinfo = searchRelation(neo4jconn,node,direction);
relinfo.Relations.RelationData{1}
```

```
ans = struct with fields:
    StartDate: '01/01/2018'
```

Close the database connection.

```
close(neo4jconn)
```

Set Multiple Relationship Properties and Return Output

Set relationship properties for multiple relationships in a Neo4j® database. Access the updated relationship information using an output argument.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the Message property of the Neo4j connection object `neo4jconn`. The blank Message property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
    []
```

Find the origin node with the node identifier 3 and retrieve its node information.

```
nodeid = 3;
nodeinfo = searchNodeByID(neo4jconn,nodeid);
```

Find the outgoing relationships from the origin node.

```
direction = "out";
relinfo = searchRelation(neo4jconn,nodeinfo,direction);
relation = relinfo.Relations.RelationObject;
```

Set the `MeetLocation` property for two outgoing relationships to different values using the Neo4j database connection. Create a structure array to store the properties. The `relationinfo` output argument is a `Neo4jRelation` object.

```
properties(1).MeetLocation = "Chicago";
properties(2).MeetLocation = "Miami";
relationinfo = setRelationProperty(neo4jconn,relation,properties);
```

Display the relationship information for the two updated relationships.

```
relationinfo.RelationData{1:2}
```

```
ans = struct with fields:
    MeetLocation: 'Chicago'
```

```
ans = struct with fields:
    MeetLocation: 'Miami'
```

Close the database connection.

```
close(neo4jconn)
```

Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a `Neo4jConnect` object created with the function `neo4j`.

relation — Relationship

Neo4jRelation object | Neo4jRelation object array | numeric scalar | numeric vector

Relationship in a Neo4j database, specified as a `Neo4jRelation` object, `Neo4jRelation` object array, numeric scalar, or numeric vector. For a single relationship, use a `Neo4jRelation` object or a numeric scalar that contains the relationship identifier. For multiple relationships, use a `Neo4jRelation` object array or a numeric vector that contains an array of relationship identifiers.

Example: 15

Example: [15,16,17]

properties — Relationship properties

structure | structure array | table

Relationship properties, specified as a structure, structure array, or table.

If a property does not exist, then the `setRelationProperty` function adds a new property. If the property exists, then the function sets a new value for the existing property.

When you specify a structure, the `setRelationProperty` function converts each field and its corresponding value to a property and its corresponding value in the database relationship. When you specify a table that contains one row, the function converts each variable and its corresponding value to a property and its corresponding value in the database relationship.

Specify a structure array or a table with multiple rows to update multiple relationships in the database.

The dimensions of the data in the structure array or table must be the same as the number of the specified relationships in the database to update. However, you can use a scalar structure to set the same values for multiple relationships in the database simultaneously.

Data Types: `struct` | `table`

Output Arguments

relationinfo — Relationship information

Neo4jRelation object | table

Relationship information, returned as a Neo4jRelation object for one relationship or as a table for multiple relationships.

For multiple relationships, the table contains these variables:

- `StartNodeID` — Node identifier of the start node for each matched relationship
- `RelationType` — Character vector that denotes the relationship type for each matched relationship
- `EndNodeID` — Node identifier of the end node for each matched relationship
- `RelationData` — Structure array that contains property keys associated with each matched relationship
- `RelationObject` — Neo4jRelation object for each matched relationship

The row names in the table are Neo4j relationship identifiers.

Version History

Introduced in R2019a

See Also

`neo4j` | `removeRelationProperty` | `setNodeProperty` | `removeNodeProperty` | `close` | `searchNode` | `searchNodeByID` | `searchRelation`

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

removeRelationProperty

Package: database.neo4j.http

Remove properties from relationships in Neo4j database

Syntax

```
removeRelationProperty(neo4jconn, relation, propertyNames)
relationinfo = removeRelationProperty(neo4jconn, relation, propertyNames)
```

Description

`removeRelationProperty(neo4jconn, relation, propertyNames)` removes properties from one or more relationships in a Neo4j database using a Neo4j database connection.

`relationinfo = removeRelationProperty(neo4jconn, relation, propertyNames)` returns updated relationship information as a `Neo4jRelation` object for one relationship or as a table for multiple relationships.

Examples

Remove One Relationship Property

Remove one property from a single relationship in a Neo4j® database and access the relationship.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url, username, password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

Retrieve the second node in the table of node information. This node has the label `Person`.

```
nlabel = "Person";
nodeinfo = searchNode(neo4jconn, nlabel);
node = nodeinfo.NodeObject(2);
```

Find the outgoing relationship from the origin node.

```
direction = "out";
relinfo = searchRelation(neo4jconn,node,direction);
relation = relinfo.Relations.RelationObject;
```

Set the `StartDate` property for a single relationship in the database using the Neo4j database connection.

```
properties.StartDate = "01/01/2018";
setRelationProperty(neo4jconn,relation,properties)
```

Display the relationship information for the updated relationship.

```
relinfo = searchRelation(neo4jconn,node,direction);
relinfo.Relations.RelationData{1}
```

```
ans = struct with fields:
    StartDate: '01/01/2018'
```

Remove the relationship property.

```
propertyNames = "StartDate";
removeRelationProperty(neo4jconn,relation,propertyNames)
```

Display the relationship information for the updated relationship.

```
relinfo = searchRelation(neo4jconn,node,direction);
relinfo.Relations.RelationData{1}
```

```
ans = struct with no fields.
```

Close the database connection.

```
close(neo4jconn)
```

Remove Multiple Relationship Properties and Return Output

Remove relationship properties from multiple relationships in a Neo4j® database. Access the updated relationship information using an output argument.

Create a Neo4j database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';
neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j connection object `neo4jconn`. The blank `Message` property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =  
    []
```

Find the origin node with the node identifier 3 and retrieve its node information.

```
nodeid = 3;  
nodeinfo = searchNodeByID(neo4jconn,nodeid);
```

Find the outgoing relationships from the origin node.

```
direction = "out";  
relinfo = searchRelation(neo4jconn,nodeinfo,direction);  
relation = relinfo.Relations.RelationObject;
```

Set the `StartDate` and `EndDate` properties for the outgoing relationships using the Neo4j database connection.

```
properties.StartDate = "01/01/2018";  
properties.EndDate = "12/31/2018";  
relationinfo = setRelationProperty(neo4jconn,relation,properties);
```

Display the relationship information for the two updated relationships.

```
relationinfo.RelationData{1:2}
```

```
ans = struct with fields:  
  StartDate: '01/01/2018'  
  EndDate: '12/31/2018'
```

```
ans = struct with fields:  
  StartDate: '01/01/2018'  
  EndDate: '12/31/2018'
```

Remove the relationship properties using the property names. The `relationinfo` output argument is a `Neo4jRelation` object.

```
propertyNames = ["StartDate" "EndDate"];  
relationinfo = removeRelationProperty(neo4jconn,relation,propertyNames);
```

Display the relationship information for the two updated relationships.

```
relationinfo.RelationData{1:2}
```

```
ans = struct with no fields.
```

```
ans = struct with no fields.
```

Close the database connection.

```
close(neo4jconn)
```


Input Arguments

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, specified as a Neo4jConnect object created with the function neo4j.

relation — Relationship

Neo4jRelation object | Neo4jRelation object array | numeric scalar | numeric vector

Relationship in a Neo4j database, specified as a Neo4jRelation object, Neo4jRelation object array, numeric scalar, or numeric vector. For a single relationship, use a Neo4jRelation object or a numeric scalar that contains the relationship identifier. For multiple relationships, use a Neo4jRelation object array or a numeric vector that contains an array of relationship identifiers.

Example: 15

Example: [15,16,17]

propertyNames — Property names

character vector | cell array of character vectors | string scalar | string array

Property names, specified as a character vector, cell array of character vectors, string scalar, or string array. For one property, use a character vector or string scalar. For multiple properties, use a cell array of character vectors or a string array.

Example: "Analyst"

Example: ["Analyst" "Clerk"]

Data Types: char | string

Output Arguments

relationinfo — Relationship information

Neo4jRelation object | table

Relationship information, returned as a Neo4jRelation object for one relationship or as a table for multiple relationships.

For multiple relationships, the table contains these variables:

- **StartNodeID** — Node identifier of the start node for each matched relationship
- **RelationType** — Character vector that denotes the relationship type for each matched relationship
- **EndNodeID** — Node identifier of the end node for each matched relationship
- **RelationData** — Structure array that contains property keys associated with each matched relationship
- **RelationObject** — Neo4jRelation object for each matched relationship

The row names in the table are Neo4j relationship identifiers.

Version History

Introduced in R2019a

See Also

neo4j | setRelationProperty | setNodeProperty | removeNodeProperty | close | searchNode | searchNodeByID | searchRelation

Topics

“Update Friend Information in Social Neighborhood” on page 9-11

“Graph Database Workflow for Neo4j Database Interfaces” on page 9-6

connection

MongoDB C++ interface connection

Description

The `connection` object enables you to connect to MongoDB stored on one or more database servers. Using the `connection` object, you can manage document collections in the database. You can also query documents stored in a collection and import them into the MATLAB workspace. From MATLAB, you can export MATLAB tables, structures, and objects into MongoDB. For details about MongoDB, see the MongoDB Manual.

Creation

Create the `connection` object using the `mongoc` function.

Properties

Database — Database name

character vector

Database name, specified as a character vector.

The `dbname` input argument of the `mongoc` function sets this property.

To change the name of the database, use dot notation to set this property; for example:

```
conn.Database = "otherDatabase";
```

Example: "databasename"

Data Types: char

UserName — User name

character vector

This property is read-only.

User name, specified as a character vector.

The `UserName` name-value argument of the `mongoc` function sets this property.

Example: "username"

Data Types: char

Server — Server name

string scalar

This property is read-only.

Server name, specified as a string scalar.

The `server` input argument of the `mongoc` function sets this property.

Example: "server1"

Data Types: `string`

Port — Port number

numeric scalar | numeric vector

This property is read-only.

Port number, specified as a numeric scalar for one port or a numeric vector for multiple ports.

The `port` input argument of the `mongoc` function sets this property.

Example: 27017

Data Types: `double`

CollectionNames — Collection names

string scalar | string array

This property is read-only.

Collection names of all collections defined in MongoDB, specified as a string scalar for one collection or string array for multiple collections.

Example: [13×1 `string`]

Data Types: `string`

Object Functions

MongoDB Connection

`isopen` Determine if MongoDB C++ interface connection is open

`close` Close MongoDB C++ interface connection

Import Document Collections into MATLAB

`count` Count total number of documents in MongoDB collection

`find` Retrieve documents in MongoDB collection

Export and Manage Document Collections in MongoDB

`createCollection` Create MongoDB collection

`dropCollection` Drop MongoDB collection

`insert` Insert one or multiple documents into MongoDB collection

`remove` Remove one or multiple documents from MongoDB collection

`update` Update one or multiple documents in MongoDB collection

Examples

Create MongoDB C++ Interface Connection

Connect to MongoDB® using the MongoDB C++ interface and count the total number of documents in a collection.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server,port,dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the connection object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
    1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Determine the number of documents in the `employees` collection. The collection contains seven documents.

```
collection = "employees";
n = count(conn,collection)

n = int64
    7
```

Close the MongoDB connection.

```
close(conn)
```

Version History

Introduced in R2021b

See Also

Topics

“Import and Analyze Data from MongoDB Using MongoDB C++ Interface” on page 10-2

“Import Filtered Data from MongoDB Using MongoDB C++ Interface” on page 10-4

“Import Large Data from MongoDB Using MongoDB C++ Interface” on page 10-6

“Export MATLAB Data into MongoDB Using MongoDB C++ Interface” on page 10-8

“Import and Export MATLAB Objects Using MongoDB and MongoDB C++ Interface” on page 10-10

External Websites

MongoDB Manual

mongoc

Create MongoDB C++ interface connection

Syntax

```
conn = mongoc(server, port, dbname)
conn = mongoc(server, port, dbname, Name=Value)
```

Description

`conn = mongoc(server, port, dbname)` creates a MongoDB connection using the MongoDB C++ interface with the database server, port number, and database name.

`conn = mongoc(server, port, dbname, Name=Value)` specifies additional options using one or more name-value arguments. For example, `UserName="adminuser"` specifies the user name for the connection.

Examples

Create MongoDB C++ Interface Connection

Connect to MongoDB® using the MongoDB C++ interface and count the total number of documents in a collection.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
      Database: "mongotest"
      UserName: ""
      Server: "dbtb01"
      Port: 27017
      CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is `27017`.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Determine the number of documents in the `employees` collection. The collection contains seven documents.

```
collection = "employees";
n = count(conn, collection)

n = int64
     7
```

Close the MongoDB connection.

```
close(conn)
```

Create MongoDB Connection Using User Name and Password

Connect to MongoDB using the MongoDB C++ interface and count the total number of documents in a collection. Specify a user name and password to connect to the database.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017. Specify the user name `adminuser` and password `matlab` by setting the `UserName` and `Password` name-value arguments, respectively.

```
conn = mongoc("dbtb01", 27017, "mongotest", UserName="adminuser", Password="matlab")
conn =

connection with properties:
    Database: "mongotest"
    UserName: "adminuser"
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB C++ interface connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is `adminuser`.
- The database server is `dbtb01`.
- The port number is `27017`.
- This database contains 13 document collections.

Check the MongoDB C++ interface connection.


```
isopen(conn)
```

```
ans =
```

```
    logical
```

```
    1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Determine the number of documents in the `employees` collection. The collection contains seven documents.

```
collection = "employees";  
n = count(conn, collection)
```

```
n =
```

```
    7
```

Close the MongoDB C++ interface connection.

```
close(conn)
```

Input Arguments

server — Server name

string scalar | string array

Server name, specified as a string scalar for one database server name or a string array for multiple database server names.

Example: "localhost"

Data Types: string

port — Port number

numeric scalar | numeric vector

Port number, specified as a numeric scalar for one port or a numeric vector for multiple ports.

Example: 27017

Data Types: double

dbname — Database name

string scalar

Database name, specified as a string scalar.

Example: "employeesdb"

Data Types: string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `conn = mongoc(server,port,dbname,UserName="username",Password="pwd")` creates a MongoDB C++ interface connection using the specified user name and password.

UserName — User name

string scalar

User name, specified as a string scalar. Contact your MongoDB administrator for access credentials.

If you specify the `UserName` name-value argument, then you must also specify the `Password` name-value argument.

Example: "username"

Data Types: string

Password — Password

string scalar

Password, specified as a string scalar. Contact your MongoDB administrator for access credentials.

If you specify the `Password` name-value argument, then you must also specify the `UserName` name-value argument.

Example: "pwd"

Data Types: string

Output Arguments

conn — MongoDB C++ connection

connection object

MongoDB C++ connection, returned as a connection object.

Version History

Introduced in R2021b

See Also

Objects

connection

Functions

isopen | count | find | close

Topics

"Import and Analyze Data from MongoDB Using MongoDB C++ Interface" on page 10-2

"Import Filtered Data from MongoDB Using MongoDB C++ Interface" on page 10-4

“Import Large Data from MongoDB Using MongoDB C++ Interface” on page 10-6

“Export MATLAB Data into MongoDB Using MongoDB C++ Interface” on page 10-8

“Import and Export MATLAB Objects Using MongoDB and MongoDB C++ Interface” on page 10-10

External Websites

MongoDB Manual

close

Package: `database.mongo`

Close MongoDB C++ interface connection

Syntax

```
close(conn)
```

Description

`close(conn)` closes the MongoDB C++ interface connection.

Examples

Create MongoDB C++ Interface Connection

Connect to MongoDB® using the MongoDB C++ interface and count the total number of documents in a collection.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is `27017`.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)
```

```
ans = logical
      1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Determine the number of documents in the `employees` collection. The collection contains seven documents.

```
collection = "employees";
n = count(conn, collection)

n = int64
      7
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — **MongoDB C++ interface connection**
connection object

MongoDB C++ interface connection, specified as a `connection` object.

Version History

Introduced in R2021b

See Also

Objects
connection

Functions
mongoc | isopen | count | find

Topics

"Import and Analyze Data from MongoDB Using MongoDB C++ Interface" on page 10-2

"Import Filtered Data from MongoDB Using MongoDB C++ Interface" on page 10-4

"Import Large Data from MongoDB Using MongoDB C++ Interface" on page 10-6

External Websites

MongoDB Manual

count

Package: `database.mongo`

Count total number of documents in MongoDB collection

Syntax

```
n = count(conn, collection)
n = count(conn, collection, Query=mongoquery)
```

Description

`n = count(conn, collection)` returns the total number of documents in a collection by using the MongoDB C++ interface connection.

`n = count(conn, collection, Query=mongoquery)` returns the total number of documents in an executed MongoDB query on a collection.

Examples

Count Documents in Collection

Connect to MongoDB® using the MongoDB C++ interface and count the total number of documents in a collection.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is `27017`.

- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Determine the number of documents in the `employees` collection. The collection contains seven documents.

```
collection = "employees";
n = count(conn, collection)

n = int64
     7
```

Close the MongoDB connection.

```
close(conn)
```

Count Documents in MongoDB Query

Connect to MongoDB® using the MongoDB C++ interface and count the total number of documents in a MongoDB query on a collection in the database. Here, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
      Database: "mongotest"
      UserName: ""
      Server: "dbtb01"
      Port: 27017
      CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.

- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)
ans = logical
      1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Create a JSON-style query as a string scalar that contains a JSON-style string. This query sets the department identifier field equal to 80.

```
mongoquery = "{\"department_id\":80}";
```

Use the MongoDB query on the `employees` collection to count the total number of employees who work in the specified department. A total of four employees work in the department.

```
collection = "employees";
n = count(conn, collection, Query=mongoquery)

n = int64
    4
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB C++ interface connection

connection object

MongoDB C++ interface connection, specified as a connection object.

collection — Collection name

string scalar

Collection name, specified as a string scalar.

Example: "taxidata"

Data Types: string

mongoquery — MongoDB query

string scalar | character vector

MongoDB query, specified as a string scalar or character vector. Specify a JSON-style string to query the database.

Example: "{\"department\":\"Sales\"}" queries the database for documents where the department field is equal to Sales.

Example: `{"salary": {"$gt": 90000}}` queries the database for documents where the value of the `salary` field is greater than 90000.

Data Types: `string` | `char`

Output Arguments

n — Total number of documents

`int64` scalar

Total number of documents in a MongoDB collection or query, returned as an `int64` scalar.

Version History

Introduced in R2021b

See Also

`mongoc` | `isopen` | `find` | `close`

Topics

“Import and Analyze Data from MongoDB Using MongoDB C++ Interface” on page 10-2

“Import Filtered Data from MongoDB Using MongoDB C++ Interface” on page 10-4

“Import Large Data from MongoDB Using MongoDB C++ Interface” on page 10-6

External Websites

MongoDB Manual

createCollection

Package: database.mongo

Create MongoDB collection

Syntax

```
createCollection(conn, collection)
```

Description

`createCollection(conn, collection)` creates a collection in MongoDB by using the MongoDB C++ interface connection.

Examples

Create Collection in MongoDB

Connect to MongoDB® using the MongoDB C++ interface and create a collection.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
  Database: "mongotest"
  UserName: ""
  Server: "dbtb01"
  Port: 27017
  CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)
```

```
ans = logical
      1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Create a collection in the database using the MongoDB connection. Specify the collection name `taxidata`.

```
collection = "taxidata";
createCollection(conn, collection)
```

Display the collections in the database by using the `CollectionNames` property. The database contains the new collection `taxidata`.

```
conn.CollectionNames
```

```
ans = 14x1 string
      "dateissue"
      "product"
      "timestamps"
      "restaurants"
      "employees"
      "bsontest"
      "taxidata"
      "airlinesmall"
      "largedata"
      "patients"
      "genderMeanAge"
      "escalation"
      "nyctaxi"
      "tsunamis"
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB C++ interface connection

connection object

MongoDB C++ interface connection, specified as a connection object.

collection — Collection name

string scalar

Collection name, specified as a string scalar.

Example: "taxidata"

Data Types: string

Version History

Introduced in R2021b

See Also

`mongoc` | `isopen` | `find` | `dropCollection` | `close`

Topics

“Import and Analyze Data from MongoDB Using MongoDB C++ Interface” on page 10-2

“Import Filtered Data from MongoDB Using MongoDB C++ Interface” on page 10-4

“Import Large Data from MongoDB Using MongoDB C++ Interface” on page 10-6

External Websites

MongoDB Manual

dropCollection

Package: database.mongo

Drop MongoDB collection

Syntax

```
dropCollection(conn, collection)
```

Description

`dropCollection(conn, collection)` drops an existing collection from MongoDB by using the MongoDB C++ interface connection.

Examples

Drop Collection in MongoDB

Connect to MongoDB® using the MongoDB C++ interface and drop an existing collection.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
  Database: "mongotest"
  UserName: ""
  Server: "dbtb01"
  Port: 27017
  CollectionNames: [14x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is `27017`.
- This database contains 14 document collections.

Verify the MongoDB connection.

```
isopen(conn)
```

```
ans = logical
      1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Display the collections in the database before dropping a collection by using the `CollectionNames` property.

```
conn.CollectionNames
```

```
ans = 14x1 string
      "dateissue"
      "product"
      "timestamps"
      "restaurants"
      "employees"
      "bsontest"
      "taxidata"
      "airlinesmall"
      "largedata"
      "patients"
      "genderMeanAge"
      "escalation"
      "nyctaxi"
      "tsunamis"
```

Drop an existing collection from the database by using the MongoDB connection. Specify the collection name `taxidata`.

```
collection = "taxidata";
dropCollection(conn, collection)
```

Display the collections in the database again by using the `CollectionNames` property. The database no longer contains the collection `taxidata`.

```
conn.CollectionNames
```

```
ans = 13x1 string
      "dateissue"
      "product"
      "timestamps"
      "restaurants"
      "employees"
      "bsontest"
      "airlinesmall"
      "largedata"
      "patients"
      "genderMeanAge"
      "escalation"
      "nyctaxi"
      "tsunamis"
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn – MongoDB C++ interface connection

connection object

MongoDB C++ interface connection, specified as a connection object.

collection – Collection name

string scalar

Collection name, specified as a string scalar.

Example: "taxidata"

Data Types: string

Version History

Introduced in R2021b

See Also

`mongoc` | `isopen` | `find` | `createCollection` | `close`

Topics

"Import and Analyze Data from MongoDB Using MongoDB C++ Interface" on page 10-2

"Import Filtered Data from MongoDB Using MongoDB C++ Interface" on page 10-4

"Import Large Data from MongoDB Using MongoDB C++ Interface" on page 10-6

External Websites

MongoDB Manual

find

Package: database.mongo

Retrieve documents in MongoDB collection

Syntax

```
documents = find(conn, collection)
documents = find(conn, collection, Name=Value)
```

Description

`documents = find(conn, collection)` returns all documents in a collection by using the MongoDB C++ interface connection.

`documents = find(conn, collection, Name=Value)` specifies additional options using one or more name-value arguments. For example, `Limit=10` limits the number of documents returned to 10.

Examples

Retrieve All Documents in Collection

Connect to MongoDB® using the MongoDB C++ interface, retrieve all documents in a collection, and import them into MATLAB®. In this example, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)
```

```
conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the connection object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.

- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)
ans = logical
      1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Specify the `employees` collection for document retrieval. Retrieve all documents in the collection by using the MongoDB connection. `documents` is a cell array of structures.

```
collection = "employees";
documents = find(conn, collection);
```

Display the first document in the collection. Each document is a structure.

```
documents{1}
ans = struct with fields:
      _id: '5d8ccb9c961c96252819ea63'
  employee_id: 100
   first_name: 'Steven'
   last_name: 'King'
      email: 'SKING'
  phone_number: '515.123.4567'
   hire_date: '2003-06-17 00:00:00.0'
      job_id: 'AD_PRES'
      salary: 24000
  department_id: 90
   temporary: 0
```

The structure fields are:

- Unique identifier
- Employee identifier
- First name
- Last name
- Email
- Phone number
- Hire date
- Job name
- Employee salary
- Department identifier
- Temporary flag

Close the MongoDB connection.

```
close(conn)
```

Retrieve All Documents in MongoDB Query

Connect to MongoDB® using the MongoDB C++ interface, retrieve all documents in a MongoDB query on a collection in the database, and import them into MATLAB®. In this example, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongoc(server,port,dbname)  
  
conn = connection with properties:  
    Database: "mongotest"  
    UserName: ""  
    Server: "dbtb01"  
    Port: 27017  
    CollectionNames: [13×1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)  
  
ans = logical  
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Specify the `employees` collection for document retrieval. Create the MongoDB query as a string scalar that contains a JSON-style string. This query retrieves all employees in the department that has the department identifier 80.

```
collection = "employees";  
mongoquery =>{"department_id":80};
```

Retrieve all documents in the MongoDB query on the `employees` collection by using the MongoDB connection. `documents` is a cell array that contains a structure for each document returned by the query.

```
documents = find(conn,collection,Query=mongoquery);
```

Close the MongoDB connection.

```
close(conn)
```

Sort Retrieved Documents in Collection

Connect to MongoDB® using the MongoDB C++ interface and retrieve documents in a MongoDB query on a collection in the database. Then, sort the results by a field in the documents. In this example, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server,port,dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Specify the `employees` collection for document retrieval. Create the MongoDB query as a string scalar that contains a JSON-style string. This query retrieves all employees in the department that has the department identifier 80.

```
collection = "employees";
mongoquery = "{"department_id":80}";
```

Create the sort query as a string scalar that contains a JSON-style string. Sort the documents by the salary field.

```
sortquery = {"salary":1.0};
```

Retrieve all documents in the MongoDB query on the `employees` collection by using the MongoDB connection, and sort the documents. `documents` is a cell array that contains a structure for each document returned by the query. The documents are sorted by salary in increasing order.

```
documents = find(conn, collection, Query=mongoquery, Sort=sortquery);
```

Display the data for the first two employees sorted by salary. The salary for the second employee is higher than the first employee.

```
documents{1:2}
```

```
ans = struct with fields:
    _id: '5d8ccc16961c96252819ea68'
    employee_id: 148
    first_name: 'Gerald'
    last_name: 'Cambrault'
    email: 'GCAMBRAU'
    phone_number: '011.44.1344.619268'
    hire_date: '2007-10-15 00:00:00.0'
    job_id: 'SA_MAN'
    salary: 11000
    commission_pct: 0.3000
    manager_id: 100
    department_id: 80
    temporary: 0
```

```
ans = struct with fields:
    _id: '5d8ccc12961c96252819ea67'
    employee_id: 147
    first_name: 'Alberto'
    last_name: 'Errazuriz'
    email: 'AERRAZUR'
    phone_number: '011.44.1344.429278'
    hire_date: '2005-03-10 00:00:00.0'
    job_id: 'SA_MAN'
    salary: 12000
    commission_pct: 0.3000
    manager_id: 100
    department_id: 80
    temporary: 0
```

Close the MongoDB connection.

```
close(conn)
```

Retrieve Specific Fields in Collection

Connect to MongoDB® using the MongoDB C++ interface and retrieve all documents in a collection. Specify the fields to retrieve for each document. Import the documents into MATLAB®. In this example, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server,port,dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Specify the `employees` collection for document retrieval. Specify the fields to retrieve for each document by using a string scalar that contains a JSON-style string. For this example, return the `department_id` and `salary` fields.

```
collection = "employees";
fields = "{\"department_id\":1.0,\"salary\":1.0}";
```

Retrieve all documents in the collection. Use the name-value argument `Projection` to retrieve the specified fields for each document. `documents` is a cell array.

```
documents = find(conn,collection,Projection=fields);
```

Display the first document in the results. In addition to the unique identifier for the document, the document contains only the specified fields.

```
documents{1}
ans = struct with fields:
    _id: '5d8ccb9c961c96252819ea63'
    salary: 24000
    department_id: 90
```

Close the MongoDB connection.

```
close(conn)
```

Retrieve Specific Number of Documents Using Offset

Connect to MongoDB® using the MongoDB C++ interface and retrieve a specific number of documents in a collection in the database. Return documents from a specific position in the results using an offset value. Import the documents into MATLAB®. In this example, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server,port,dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the connection object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)
ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Specify the `employees` collection for document retrieval.

```
collection = "employees";
```

Use the name-value argument `Skip` to skip the first five documents in the collection. Then, use the name-value argument `Limit` to return the next 10 documents in the collection. `documents` is a cell array that contains 10 documents.

```
documents = find(conn,collection,Skip=5,Limit=10);
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB C++ interface connection

connection object

MongoDB C++ interface connection, specified as a connection object.

collection — Collection name

string scalar

Collection name, specified as a string scalar.

Example: "taxidata"

Data Types: string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Skip=5, Limit=10` skips the first five documents in a collection and returns the next 10 documents.

Query — MongoDB query

string scalar | character vector

MongoDB query, specified as a string scalar or character vector. Specify a JSON-style string to query the database.

Example: `Query="{\"department\":\"Sales\"}"` queries the database for documents where the `department` field is equal to `Sales`.

Example: `Query="{\"salary\": {\"$gt\": 90000}}"` queries the database for documents where the value of the `salary` field is greater than 90000.

Example: `Query="{\"_id\":{\"$oid\":\"593fec95b78dc311e01e9204\"}}"` queries the database for the document that has the identifier `593fec95b78dc311e01e9204`.

Data Types: char | string

Projection — Fields

string scalar | character vector

Fields to retrieve in each document, specified as a string scalar or character vector. Specify a JSON-style string to describe the fields.

Example: `Projection="{\"department\":1.0,\"salary\":1.0}"` returns the department and salary fields.

Data Types: char | string

Sort — Sort field

string scalar | character vector

Sort field for documents, specified as a string scalar or character vector. Specify a JSON-style string to describe the sort field.

Example: `Sort="{\"department\":1.0}"` sorts the returned documents by the department field.

Data Types: char | string

Skip — Offset

numeric scalar

Offset from the beginning of the returned documents, specified as a numeric scalar.

Example: `Skip=5` skips the first five returned documents.

Data Types: double

Limit — Number of documents to return

numeric scalar

Number of documents to return, specified as a numeric scalar.

Example: `Limit=10` returns 10 documents.

Data Types: double

Output Arguments

documents — Documents

structure | structure array | cell array of structures

Documents in a MongoDB collection or query on a collection, returned as a structure, structure array, or cell array of structures.

Each JSON-style document is represented as a structure. The `find` function returns a:

- Structure for one document
- Structure array for multiple documents containing the same fields
- Cell array of structures for multiple documents containing different fields

Version History

Introduced in R2021b

See Also

`mongoc` | `isopen` | `insert` | `remove` | `update` | `close`

Topics

“Import and Analyze Data from MongoDB Using MongoDB C++ Interface” on page 10-2

“Import Filtered Data from MongoDB Using MongoDB C++ Interface” on page 10-4

“Import Large Data from MongoDB Using MongoDB C++ Interface” on page 10-6

External Websites

MongoDB Manual

insert

Package: database.mongo

Insert one or multiple documents into MongoDB collection

Syntax

```
n = insert(conn, collection, documents)
```

Description

`n = insert(conn, collection, documents)` returns the number of documents inserted into a collection using the MongoDB C++ interface connection. Specify one or multiple documents to insert.

Examples

Insert One Document into Collection as Structure

Connect to MongoDB® using the MongoDB C++ interface and export one document from MATLAB® and insert it into a collection. Specify the document to insert as a structure. In this example, the collection represents employee data.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Create one document as the `document` structure with three fields. Set the `employee_id` field to 28, `department_id` field to 80, and `salary` field to 200,000.

```
document.employee_id = 28;
document.department_id = 80;
document.salary = 200000;
```

Specify the `employees` collection. Insert the document into the collection by using the MongoDB C++ interface connection. The `insert` function inserts one document into the collection.

```
collection = "employees";
n = insert(conn, collection, document)

n = int64
     1
```

Close the MongoDB connection.

```
close(conn)
```

Insert Multiple Documents into Collection as Structure Array

Connect to MongoDB® using the MongoDB C++ interface and export multiple documents from MATLAB® and insert them into a collection. Specify documents to insert as a structure array. In this example, the collection represents employee data.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.

- The user name is blank.
- The database server is `dbtb01`.
- The port number is `27017`.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Create two documents as structures with these fields: `employee_id`, `department_id`, and `salary`. For the `employee1` structure, set the `employee_id` field to 26, `department_id` field to 80, and `salary` field to 100,000. For the `employee2` structure, set the same fields to the values 27, 90, and 150,000 respectively. Create the `documents` structure array from these documents.

```
employee1.employee_id = 26;
employee1.department_id = 80;
employee1.salary = 100000;
```

```
employee2.employee_id = 27;
employee2.department_id = 90;
employee2.salary = 150000;
```

```
documents = [employee1 employee2];
```

Specify the `employees` collection. Insert documents into the collection using the MongoDB connection. The `insert` function inserts two documents into the collection.

```
collection = "employees";
n = insert(conn, collection, documents)

n = int64
     2
```

Close the MongoDB connection.

```
close(conn)
```

Insert Multiple Documents into Collection as Table

Connect to MongoDB® using the MongoDB C++ interface and export documents from MATLAB® and insert them into a collection. Specify documents to insert as a table. In this example, the collection represents employee data.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
```

```

dbname = "mongotest";
conn = mongoc(server,port,dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]

```

`conn` is the connection object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is `27017`.
- This database contains 13 document collections.

Verify the MongoDB connection.

```

isopen(conn)

ans = logical
     1

```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Create two documents using these workspace variables:

- `department_ids` — Double array
- `employee_ids` — Double array
- `salaries` — Double array

Create the `documents` table from these workspace variables.

```

department_ids = [80;90];
employee_ids = [24;25];
salaries = [100000;150000];
documents = table(department_ids,employee_ids,salaries);

```

Specify the `employees` collection. Insert documents into the collection using the MongoDB connection. The `insert` function inserts two documents into the collection.

```

collection = "employees";
n = insert(conn,collection,documents)

n = int64
     2

```

Close the MongoDB connection.

```

close(conn)

```

Insert Multiple Documents into Collection as Cell Array of Structures

Connect to MongoDB® using the MongoDB C++ interface and export documents from MATLAB® and insert them into a collection. Specify documents to insert as a cell array of structures. In this example, the collection represents employee data.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server,port,dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Create two documents as the structures `employee1` and `employee2`. Create the documents cell array using these structures.

```
employee1.department_id = 90;
employee1.employee_id = 22;
employee1.salary = 100000;

employee2.department_id = 80;
employee2.employee_id = 23;
employee2.salary = 150000;

documents = {employee1;employee2};
```

Specify the `employees` collection. Insert documents into the collection using the MongoDB C++ interface connection. The `insert` function inserts two documents into the collection.

```
collection = "employees";
n = insert(conn, collection, documents)

n = int64
    2
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB C++ interface connection

connection object

MongoDB C++ interface connection, specified as a connection object.

collection — Collection name

string scalar

Collection name, specified as a string scalar.

Example: "taxidata"

Data Types: string

documents — Documents to insert

string scalar | character vector | structure | ...

Documents to insert into a MongoDB collection, specified as one of these types:

- String scalar
- Character vector
- Structure
- Structure array
- Cell array of structures
- Table
- Handle or value classes

When working with string scalars and character vectors, you specify key-value pairs as shown in these examples.

- String scalar — '{"department":"Sales","employeename":"George Mason"}'
- Character vector — {'department':'Sales','employeename':'George Mason'}

For handle and value classes, you can define your own class. After you instantiate a class, you can insert the resulting object into MongoDB. However, the resulting object properties must contain data types that can be converted to MATLAB data types. For example, if one of the object properties is a

Java object, then you cannot insert the object into MongoDB. For details about these classes, see “Handle Classes”.

Output Arguments

n — Number of documents inserted

`int64` scalar

Number of documents inserted into a collection in the database, returned as an `int64` scalar.

Note The `insert` function does not return the `containers.Map` data type.

Version History

Introduced in R2021b

See Also

`mongoc` | `isopen` | `remove` | `update` | `close`

Topics

“Export MATLAB Data into MongoDB Using MongoDB C++ Interface” on page 10-8

“Import and Export MATLAB Objects Using MongoDB and MongoDB C++ Interface” on page 10-10

External Websites

MongoDB Manual

isopen

Package: database.mongo

Determine if MongoDB C++ interface connection is open

Syntax

```
i = isopen(conn)
```

Description

`i = isopen(conn)` returns 1 if the MongoDB connection is open and 0 if it is closed.

Examples

Verify MongoDB C++ Interface Connection

Connect to MongoDB® using the MongoDB C++ interface and count the total number of documents in a collection. Then, verify the connection is closed.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
  Database: "mongotest"
  UserName: ""
  Server: "dbtb01"
  Port: 27017
  CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.

```
isopen(conn)
```

```
ans = logical  
    1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Determine the number of documents in the `employees` collection. The collection contains seven documents.

```
collection = "employees";  
n = count(conn, collection)
```

```
n = int64  
    7
```

Close the MongoDB connection.

```
close(conn)
```

Verify the MongoDB connection is closed.

```
isopen(conn)
```

```
ans = logical  
    0
```

Input Arguments

conn — MongoDB C++ interface connection

connection object

MongoDB C++ interface connection, specified as a connection object.

Version History

Introduced in R2021b

See Also

Objects

connection

Functions

mongoc | count | find | close

Topics

“Import and Analyze Data from MongoDB Using MongoDB C++ Interface” on page 10-2

“Import Filtered Data from MongoDB Using MongoDB C++ Interface” on page 10-4

“Import Large Data from MongoDB Using MongoDB C++ Interface” on page 10-6

External Websites

MongoDB Manual

remove

Package: database.mongo

Remove one or multiple documents from MongoDB collection

Syntax

```
n = remove(conn, collection, mongoquery)
```

Description

`n = remove(conn, collection, mongoquery)` returns the number of documents removed from a collection using the MongoDB C++ interface connection. Use a MongoDB query to specify removing one or multiple documents.

Examples

Remove Documents Using MongoDB Query

Connect to MongoDB® using the MongoDB C++ interface and remove documents from a collection. Specify a MongoDB query to determine which documents to remove. In this example, the collection represents employee data.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [14x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 14 document collections.

Verify the MongoDB connection.

```
isopen(conn)

ans = logical
     1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Create a MongoDB query to identify documents to remove. For this example, specify the `employeedata` collection. Create the MongoDB query to identify documents in the department that has the department identifier set to 80.

```
collection = "employeedata";
mongoquery = {"department_id":80};
```

Remove documents using the MongoDB query. The `remove` function removes three documents from the collection.

```
n = remove(conn, collection, mongoquery)

n = int64
     3
```

Close the MongoDB connection.

```
close(conn)
```

Remove All Documents from Collection

Connect to MongoDB® using the MongoDB C++ interface and remove all documents from a collection. In this example, the collection represents employee data.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
      Database: "mongotest"
      UserName: ""
      Server: "dbtb01"
      Port: 27017
      CollectionNames: [14x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.

- The database server is `dbtb01`.
- The port number is `27017`.
- This database contains 14 document collections.

Verify the MongoDB connection.

```
isopen(conn)
ans = logical
      1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Remove all documents from the `employeedata` collection. Use an empty MongoDB query to specify removing all documents. The `remove` function removes three documents from the collection.

```
collection = "employeedata";
n = remove(conn, collection, "{}")
n = int64
     3
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB C++ interface connection

connection object

MongoDB C++ interface connection, specified as a `connection` object.

collection — Collection name

string scalar

Collection name, specified as a string scalar.

Example: `"taxidata"`

Data Types: `string`

mongoquery — MongoDB query

string scalar | character vector

MongoDB query, specified as a string scalar or character vector. Specify a JSON-style string to query the database.

Example: `"{'department': 'Sales'}"` queries the database for documents where the `department` field is equal to `Sales`.

Example: `"{'salary': {'$gt': 90000}}"` queries the database for documents where the value of the `salary` field is greater than `90000`.

Data Types: `string` | `char`

Output Arguments

n — Number of documents removed

numeric scalar

Number of documents removed from a collection in the database, returned as a numeric scalar.

Version History

Introduced in R2021b

See Also

mongoc | isopen | insert | update | close

Topics

“Export MATLAB Data into MongoDB Using MongoDB C++ Interface” on page 10-8

“Import and Export MATLAB Objects Using MongoDB and MongoDB C++ Interface” on page 10-10

External Websites

MongoDB Manual

update

Package: `database.mongo`

Update one or multiple documents in MongoDB collection

Syntax

```
n = update(conn, collection, findquery, updatequery)
```

Description

`n = update(conn, collection, findquery, updatequery)` returns the number of documents updated in a collection using the MongoDB C++ interface connection. Use MongoDB queries to find and update documents.

Examples

Update Documents in Collection

Connect to MongoDB® using the MongoDB C++ interface and update documents in a collection. Find documents to update by using a MongoDB query. Specify the criteria for the update by using a MongoDB query. In this example, the collection represents employee data.

Create a MongoDB connection to the database `mongotest` using the MongoDB C++ interface. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongoc(server, port, dbname)

conn = connection with properties:
    Database: "mongotest"
    UserName: ""
    Server: "dbtb01"
    Port: 27017
    CollectionNames: [13x1 string]
```

`conn` is the `connection` object that contains the MongoDB connection. The object properties contain information about the connection and the database.

- The database name is `mongotest`.
- The user name is blank.
- The database server is `dbtb01`.
- The port number is 27017.
- This database contains 13 document collections.

Verify the MongoDB connection.


```
isopen(conn)
ans = logical
      1
```

The database connection is successful because the `isopen` function returns 1. Otherwise, the database connection is closed.

Specify the `employees` collection. Create a MongoDB query to find employees in the department where the department identifier is set to `90`. Then, create a MongoDB query to increase the value in the salary field by `5000`.

```
collection = "employees";
findquery = "{\"department_id\":90}";
updatequery = "{\"$inc\":{\"salary\":5000}}";
```

Increase the salaries for all employees in the department using the MongoDB connection. The `update` function updates four documents in the collection.

```
n = update(conn,collection,findquery,updatequery)
n = int64
      4
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB C++ interface connection

connection object

MongoDB C++ interface connection, specified as a connection object.

collection — Collection name

string scalar

Collection name, specified as a string scalar.

Example: "taxidata"

Data Types: string

findquery — MongoDB find query

string scalar | character vector

MongoDB find query, specified as a string scalar or character vector. Specify a JSON-style string to find documents in the database.

Example: "{\"department\":\"Sales\"}" finds all documents where the department field is equal to Sales.

Example: "{\"_id\":{\"\$oid\":\"593fec95b78dc311e01e9204\"}}}" finds the document that has the identifier 593fec95b78dc311e01e9204.

Data Types: char | string

updatequery — MongoDB update query

string scalar | character vector

MongoDB update query, specified as a string scalar or character vector. Use a JSON-style string to specify the criteria for the update.

Example: `"{"$inc":{"salary":5000}}"` increases the values in the salary field by 5000.

Data Types: char | string

Output Arguments**n — Number of documents updated**

numeric scalar

Number of documents updated in a collection in the database, returned as a numeric scalar.

Version History**Introduced in R2021b****See Also**

mongoc | isopen | insert | remove | close

Topics

“Export MATLAB Data into MongoDB Using MongoDB C++ Interface” on page 10-8

“Import and Export MATLAB Objects Using MongoDB and MongoDB C++ Interface” on page 10-10

External Websites

MongoDB Manual

CassandraConnectionOptions

Apache Cassandra database connection options

Description

Create connection options for an Apache Cassandra database connection.

After you create an `CassandraConnectionOptions` object, set the connection options, test the connection, and save the data source, you can create an Apache Cassandra database connection using the saved data source. The connection options include the options required to make a Cassandra database connection.

Creation

To create a `CassandraConnectionOptions` object, use the `databaseConnectionOptions` function.

Properties

DataSourceName — Data source name

string scalar

Data source name, specified as a string scalar. You can use the data source name in the `apacheCassandra` function to create a Cassandra database connection.

Example: "ApacheCassandra"

Data Types: string

Vendor — Database vendor

string scalar

This property is read-only.

Database vendor, specified as a string scalar. Set this property using the `vendor` input argument in the `databaseConnectionOptions` function.

Example: "Cassandra"

Data Types: string

ContactPoints — Contact points

"localhost" (default) | string scalar | string array

Contact points that are host addresses for one node or for multiple nodes in the Cassandra cluster, specified as a string scalar or string array. Specify a string scalar for one node. Or, specify a string array for multiple nodes.

You can specify a local host or the IP address of a different machine in the Cassandra cluster.

When you specify multiple nodes, if the connection to one host fails, then the `apacheCassandra` function connects to the other nodes in the cell array or string array until a connection succeeds. If a connection attempt fails for all specified nodes, the function displays an error message. If one or more nodes are not available, enter multiple nodes in the string array to increase the likelihood of a successful connection.

Data Types: `string`

PortNumber — Port number

9042 (default) | positive numeric scalar

Port number for connection to the host, specified as a positive numeric scalar.

Data Types: `double`

SSLEnabled — SSL-enabled connection

`false` (default) | `true`

SSL-enabled connection, specified as the value `false` or `true`. Setting this argument to `true` creates an SSL-enabled connection to the Cassandra database.

Data Types: `logical`

LoginTimeout — Login timeout

5 (default) | positive numeric scalar

Login timeout, specified as a positive numeric scalar. The login timeout specifies the number of seconds that the C++ driver waits while trying to connect to the Cassandra database before throwing an error.

Data Types: `double`

RequestTimeout — Request timeout

12 (default) | positive numeric scalar

This property is read-only.

Request timeout, specified as a positive numeric scalar. The request timeout indicates the number of seconds the database waits to return a CQL query before throwing an error.

Data Types: `double`

Object Functions

<code>setoptions</code>	Set Apache Cassandra database connection options
<code>testConnection</code>	Test Apache Cassandra database connection
<code>reset</code>	Reset Apache Cassandra database connection options to defaults
<code>saveAsDataSource</code>	Save Apache Cassandra data source

Examples**Create Cassandra Data Source and Set Connection Options**

Configure an Apache™ Cassandra® database connection by creating a Cassandra data source, setting the Cassandra connection options, and saving the data source.

Create a Cassandra data source for a Cassandra database connection.

```
vendor = "Cassandra";
opts = databaseConnectionOptions("native", vendor)
```

```
opts =
  CassandraConnectionOptions with properties:
```

```
    DataSourceName: ""
    Vendor: "Cassandra"

    ContactPoints: "localhost"
    PortNumber: 9042
    SSLEnabled: false
    LoginTimeout: 5
    RequestTimeout: 12
```

opts is an `CassandraConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `ContactPoints` — Contact points
- `PortNumber` — Port number
- `SSLEnabled` — SSL-enabled connection
- `LoginTimeout` — Login timeout
- `RequestTimeout` — Request timeout

Configure the data source by setting the Cassandra connection options for the data source `CassandraDataSource`, local host address for one node in the cluster, port number 9042, SSL encryption that is disabled, login timeout of 5 seconds, and request timeout of 12 seconds.

```
opts = setoptions(opts, ...
  "DataSourceName", "CassandraDataSource", ...
  "ContactPoints", "localhost", "PortNumber", 9042, ...
  "SSLEnabled", false, "LoginTimeout", 5, ...
  "RequestTimeout", 12)
```

```
opts =
  CassandraConnectionOptions with properties:
```

```
    DataSourceName: "CassandraDataSource"
    Vendor: "Cassandra"

    ContactPoints: "localhost"
    PortNumber: 9042
    SSLEnabled: false
    LoginTimeout: 5
    RequestTimeout: 12
```

The `setoptions` function sets the `DataSourceName`, `ContactPoints`, `PortNumber`, `SSLEnabled`, `LoginTimeout`, and `RequestTimeout` properties in the `CassandraConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "";  
password = "";  
status = testConnection(opts,username,password)  
  
status = logical  
      1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

Version History

Introduced in R2021a

See Also

[apacheCassandra](#) | [databaseConnectionOptions](#) | [deleteDataSource](#)

External Websites

[Apache Cassandra](#)

[Apache Cassandra Documentation](#)

setoptions

Package: database.options.native.cassandra

Set Apache Cassandra database connection options

Syntax

```
opts = setoptions(opts,Option1,OptionValue1,...,OptionN,OptionValueN)
```

Description

`opts = setoptions(opts,Option1,OptionValue1,...,OptionN,OptionValueN)` sets Apache Cassandra database connection options using the `CassandraConnectionOptions` object `opts`.

Examples

Create Cassandra Data Source and Set Connection Options

Configure an Apache™ Cassandra® database connection by creating a Cassandra data source, setting the Cassandra connection options, and saving the data source.

Create a Cassandra data source for a Cassandra database connection.

```
vendor = "Cassandra";
opts = databaseConnectionOptions("native",vendor)
```

```
opts =
  CassandraConnectionOptions with properties:
```

```
    DataSourceName: ""
    Vendor: "Cassandra"

    ContactPoints: "localhost"
    PortNumber: 9042
    SSLEnabled: false
    LoginTimeout: 5
    RequestTimeout: 12
```

`opts` is an `CassandraConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `ContactPoints` — Contact points
- `PortNumber` — Port number
- `SSLEnabled` — SSL-enabled connection
- `LoginTimeout` — Login timeout

- RequestTimeout — Request timeout

Configure the data source by setting the Cassandra connection options for the data source `CassandraDataSource`, local host address for one node in the cluster, port number 9042, SSL encryption that is disabled, login timeout of 5 seconds, and request timeout of 12 seconds.

```
opts = setoptions(opts, ...
  "DataSourceName", "CassandraDataSource", ...
  "ContactPoints", "localhost", "PortNumber", 9042, ...
  "SSLEnabled", false, "LoginTimeout", 5, ...
  "RequestTimeout", 12)
```

```
opts =
  CassandraConnectionOptions with properties:

      DataSourceName: "CassandraDataSource"
      Vendor: "Cassandra"

      ContactPoints: "localhost"
      PortNumber: 9042
      SSLEnabled: false
      LoginTimeout: 5
      RequestTimeout: 12
```

The `setoptions` function sets the `DataSourceName`, `ContactPoints`, `PortNumber`, `SSLEnabled`, `LoginTimeout`, and `RequestTimeout` properties in the `CassandraConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)

status = logical
      1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

`CassandraConnectionOptions` object

Database connection options, specified as a `CassandraConnectionOptions` object.

Option1, OptionValue1, ..., OptionN, OptionValueN — Cassandra database connection options to set

name-value pair arguments

Cassandra database connection options to set, specified as one or more name-value pair arguments. `Option` is a character vector or string scalar that specifies the name of a Cassandra database

connection option. `OptionValue` specifies the value of the Cassandra database connection option. `OptionValue` can be a character vector, string scalar, logical scalar, or numeric scalar. You can specify any Cassandra database connection option that is a property of the `CassandraConnectionOptions` object.

Example:

```
"DataSourceName", "myDataSource", "ContactPoints", "localhost", "PortNumber", 9042
```

configures a Cassandra data source named `myDataSource` that is located on the local host address with the port number 9042.

Output Arguments

opts — Database connection options

`CassandraConnectionOptions` object

Database connection options, returned as a `CassandraConnectionOptions` object.

Version History

Introduced in R2021a

See Also

Objects

`CassandraConnectionOptions` | `connection`

Functions

`reset` | `testConnection` | `saveAsDataSource` | `databaseConnectionOptions` | `deleteDataSource`

Topics

“Modify and Delete Data Sources” on page 4-17

testConnection

Package: `database.options.native.cassandra`

Test Apache Cassandra database connection

Syntax

```
status = testConnection(opts,username,password)
[status,message] = testConnection(opts,username,password)
```

Description

`status = testConnection(opts,username,password)` tests the Apache Cassandra database connection specified by the `CassandraConnectionOptions` object `opts`, a user name, and a password.

`[status,message] = testConnection(opts,username,password)` also returns the error message associated with testing the database connection.

Examples

Create Cassandra Data Source and Set Connection Options

Configure an Apache™ Cassandra® database connection by creating a Cassandra data source, setting the Cassandra connection options, and saving the data source.

Create a Cassandra data source for a Cassandra database connection.

```
vendor = "Cassandra";
opts = databaseConnectionOptions("native",vendor)
```

```
opts =
  CassandraConnectionOptions with properties:
```

```
    DataSourceName: ""
    Vendor: "Cassandra"

    ContactPoints: "localhost"
    PortNumber: 9042
    SSLEnabled: false
    LoginTimeout: 5
    RequestTimeout: 12
```

`opts` is an `CassandraConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `ContactPoints` — Contact points

- PortNumber — Port number
- SSLEnabled — SSL-enabled connection
- LoginTimeout — Login timeout
- RequestTimeout — Request timeout

Configure the data source by setting the Cassandra connection options for the data source `CassandraDataSource`, local host address for one node in the cluster, port number 9042, SSL encryption that is disabled, login timeout of 5 seconds, and request timeout of 12 seconds.

```
opts = setoptions(opts, ...
    "DataSourceName", "CassandraDataSource", ...
    "ContactPoints", "localhost", "PortNumber", 9042, ...
    "SSLEnabled", false, "LoginTimeout", 5, ...
    "RequestTimeout", 12)
```

```
opts =
  CassandraConnectionOptions with properties:

    DataSourceName: "CassandraDataSource"
    Vendor: "Cassandra"

    ContactPoints: "localhost"
    PortNumber: 9042
    SSLEnabled: false
    LoginTimeout: 5
    RequestTimeout: 12
```

The `setoptions` function sets the `DataSourceName`, `ContactPoints`, `PortNumber`, `SSLEnabled`, `LoginTimeout`, and `RequestTimeout` properties in the `CassandraConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)

status = logical
      1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

Retrieve Message for Apache Cassandra Database Connection Test

Create and configure a Cassandra data source to a Cassandra database. Test the database connection to the Cassandra data source and retrieve the error message.

Create a Cassandra data source for a Cassandra database connection.

```

vendor = "Cassandra";
opts = databaseConnectionOptions("native",vendor)

opts =
  CassandraConnectionOptions with properties:

      DataSourceName: ""
      Vendor: "Cassandra"

      ContactPoints: "localhost"
      PortNumber: 9042
      SSLEnabled: false
      LoginTimeout: 5
      RequestTimeout: 12

```

opts is a `CassandraConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `ContactPoints` — Contact points
- `PortNumber` — Port number
- `SSLEnabled` — SSL-enabled connection
- `LoginTimeout` — Login timeout
- `RequestTimeout` — Request timeout

Configure the data source by setting the Cassandra connection options for the data source `CassandraDataSource`, local host address for one node in the cluster, port number 1500, SSL encryption that is disabled, login timeout of 5 seconds, and request timeout of 12 seconds. In this case, the port number is invalid.

```

opts = setoptions(opts, ...
  "DataSourceName", "CassandraDataSource", ...
  "ContactPoints", "localhost", "PortNumber", 1500, ...
  "SSLEnabled", false, "LoginTimeout", 5, ...
  "RequestTimeout", 12)

```

```

opts =
  CassandraConnectionOptions with properties:

      DataSourceName: "CassandraDataSource"
      Vendor: "Cassandra"

      ContactPoints: "localhost"
      PortNumber: 1500
      SSLEnabled: false
      LoginTimeout: 5
      RequestTimeout: 12

```

The `setoptions` function sets the `DataSourceName`, `ContactPoints`, `PortNumber`, `SSLEnabled`, `LoginTimeout`, and `RequestTimeout` properties in the `CassandraConnectionOptions` object.

Test the database connection using a blank user name and password. The `testConnection` function returns the logical `0`, which indicates the database connection fails because the port number is invalid. Retrieve and display the error message for the failed connection.

```
username = "";
password = "";
[status,message] = testConnection(opts,username,password)

status =

    logical

    0

message =

    'Cassandra exception: Underlying connection error: Connect error 'connection refused''
```

Input Arguments

opts — Database connection options

CassandraConnectionOptions object

Database connection options, specified as a CassandraConnectionOptions object.

username — User name

character vector | string scalar

User name required to access the database, specified as a character vector or string scalar. If no user name is required, specify an empty value `""`.

Data Types: `char` | `string`

password — Password

character vector | string scalar

Password required to access the database, specified as a character vector or string scalar. If no password is required, specify an empty value `""`.

Data Types: `char` | `string`

Output Arguments

status — Connection status

logical

Connection status, returned as a logical `true` if the connection test passes or `false` if the connection test fails.

message — Error message

character vector

Error message, returned as a character vector. If the connection test passes, then the error message is an empty character vector. Otherwise, the error message contains text from the failed database connection.

Version History

Introduced in R2021a

See Also

Objects

CassandraConnectionOptions | connection

Functions

setoptions | reset | saveAsDataSource | databaseConnectionOptions | deleteDataSource

Topics

“Modify and Delete Data Sources” on page 4-17

reset

Package: `database.options.native.cassandra`

Reset Apache Cassandra database connection options to defaults

Syntax

```
opts = reset(opts)
```

Description

`opts = reset(opts)` resets all Apache Cassandra database connection options to their default values for all properties of the `CassandraConnectionOptions` object.

Examples

Reset Cassandra Connection Options to Default Values

Create, configure, and test an Apache™ Cassandra® data source for a Cassandra database. Reset the Cassandra connection options to their default values. Then configure, test, and save the data source with different Cassandra connection options.

Create a Cassandra data source for a Cassandra database connection.

```
vendor = "Cassandra";
opts = databaseConnectionOptions("native", vendor)
```

```
opts =
  CassandraConnectionOptions with properties:
```

```
    DataSourceName: ""
    Vendor: "Cassandra"

    ContactPoints: "localhost"
    PortNumber: 9042
    SSLEnabled: false
    LoginTimeout: 5
    RequestTimeout: 12
```

`opts` is an `CassandraConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `ContactPoints` — Contact points
- `PortNumber` — Port number
- `SSLEnabled` — SSL-enabled connection
- `LoginTimeout` — Login timeout

- RequestTimeout — Request timeout

Configure the data source by setting the Cassandra connection options for the data source `CassandraDataSource`, local host address for one node in the cluster, port number 9042, SSL encryption that is disabled, login timeout of 5 seconds, and request timeout of 12 seconds.

```
opts = setoptions(opts, ...
    "DataSourceName", "CassandraDataSource", ...
    "ContactPoints", "localhost", "PortNumber", 9042, ...
    "SSLEnabled", false, "LoginTimeout", 5, ...
    "RequestTimeout", 12)
```

```
opts =
  CassandraConnectionOptions with properties:

    DataSourceName: "CassandraDataSource"
    Vendor: "Cassandra"

    ContactPoints: "localhost"
    PortNumber: 9042
    SSLEnabled: false
    LoginTimeout: 5
    RequestTimeout: 12
```

The `setoptions` function sets the `DataSourceName`, `ContactPoints`, `PortNumber`, `SSLEnabled`, `LoginTimeout`, and `RequestTimeout` properties in the `CassandraConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)
```

```
status = logical
      1
```

Reset the Cassandra connection options to their default values. The properties of the `CassandraConnectionOptions` object contain the default values.

```
opts = reset(opts)

opts =
  CassandraConnectionOptions with properties:
```

```
    DataSourceName: ""
    Vendor: "Cassandra"

    ContactPoints: "localhost"
    PortNumber: 9042
    SSLEnabled: false
    LoginTimeout: 5
    RequestTimeout: 12
```


Configure the data source by setting the Cassandra connection options for the data source `CassandraDataSource`, local host address for one node in the cluster, port number 9042, SSL encryption that is disabled, login timeout of 5 seconds, and request timeout of 20 seconds.

```
opts = setoptions(opts, ...
    "DataSourceName", "CassandraDataSource", ...
    "ContactPoints", "localhost", "PortNumber", 9042, ...
    "SSLEnabled", false, "LoginTimeout", 5, ...
    "RequestTimeout", 20)
```

```
opts =
    CassandraConnectionOptions with properties:
```

```
    DataSourceName: "CassandraDataSource"
    Vendor: "Cassandra"

    ContactPoints: "localhost"
    PortNumber: 9042
    SSLEnabled: false
    LoginTimeout: 5
    RequestTimeout: 20
```

The `setoptions` function sets the `DataSourceName`, `ContactPoints`, `PortNumber`, `SSLEnabled`, `LoginTimeout`, and `RequestTimeout` properties in the `CassandraConnectionOptions` object.

Test the database connection again.

```
username = "";
password = "";
status = testConnection(opts, username, password)
```

```
status = logical
    1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

`CassandraConnectionOptions` object

Database connection options, specified as a `CassandraConnectionOptions` object.

Output Arguments

opts — Database connection options

`CassandraConnectionOptions` object

Database connection options, returned as a `CassandraConnectionOptions` object.

Version History

Introduced in R2021a

See Also

Objects

CassandraConnectionOptions | connection

Functions

setoptions | testConnection | saveAsDataSource | databaseConnectionOptions | deleteDataSource

Topics

“Modify and Delete Data Sources” on page 4-17

saveAsDataSource

Package: database.options.native.cassandra

Save Apache Cassandra data source

Syntax

```
saveAsDataSource(opts)
```

Description

`saveAsDataSource(opts)` saves the Apache Cassandra data source specified by the `CassandraConnectionOptions` object `opts`.

Examples

Create Cassandra Data Source and Set Connection Options

Configure an Apache™ Cassandra® database connection by creating a Cassandra data source, setting the Cassandra connection options, and saving the data source.

Create a Cassandra data source for a Cassandra database connection.

```
vendor = "Cassandra";  
opts = databaseConnectionOptions("native", vendor)
```

```
opts =  
  CassandraConnectionOptions with properties:
```

```
    DataSourceName: ""  
    Vendor: "Cassandra"  
  
    ContactPoints: "localhost"  
    PortNumber: 9042  
    SSLEnabled: false  
    LoginTimeout: 5  
    RequestTimeout: 12
```

`opts` is an `CassandraConnectionOptions` object with these properties:

- `DataSourceName` — Name of the data source
- `Vendor` — Database vendor name
- `ContactPoints` — Contact points
- `PortNumber` — Port number
- `SSLEnabled` — SSL-enabled connection
- `LoginTimeout` — Login timeout
- `RequestTimeout` — Request timeout

Configure the data source by setting the Cassandra connection options for the data source `CassandraDataSource`, local host address for one node in the cluster, port number 9042, SSL encryption that is disabled, login timeout of 5 seconds, and request timeout of 12 seconds.

```
opts = setoptions(opts, ...
    "DataSourceName", "CassandraDataSource", ...
    "ContactPoints", "localhost", "PortNumber", 9042, ...
    "SSLEnabled", false, "LoginTimeout", 5, ...
    "RequestTimeout", 12)
```

```
opts =
    CassandraConnectionOptions with properties:
```

```
    DataSourceName: "CassandraDataSource"
    Vendor: "Cassandra"

    ContactPoints: "localhost"
    PortNumber: 9042
    SSLEnabled: false
    LoginTimeout: 5
    RequestTimeout: 12
```

The `setoptions` function sets the `DataSourceName`, `ContactPoints`, `PortNumber`, `SSLEnabled`, `LoginTimeout`, and `RequestTimeout` properties in the `CassandraConnectionOptions` object.

Test the database connection with a blank user name and password. The `testConnection` function returns the logical 1, which indicates the database connection is successful.

```
username = "";
password = "";
status = testConnection(opts, username, password)
```

```
status = logical
      1
```

Save the configured data source.

```
saveAsDataSource(opts)
```

Input Arguments

opts — Database connection options

`CassandraConnectionOptions` object

Database connection options, specified as a `CassandraConnectionOptions` object.

Version History

Introduced in R2021a

See Also

Objects

CassandraConnectionOptions | connection

Functions

setoptions | testConnection | reset | databaseConnectionOptions | deleteDataSource

Topics

“Modify and Delete Data Sources” on page 4-17

connection

Apache Cassandra database connection

Description

The `connection` object represents an Apache Cassandra database connection created using the Apache Cassandra database C++ interface.

After you create a `connection` object, you can use the object functions to import data from the Cassandra database into MATLAB. Or, you can export data from MATLAB to the Cassandra database. You can also explore the database structure and execute Cassandra Query Language (CQL) queries.

For details about the Cassandra database, see the Apache Cassandra Documentation.

Creation

To create a `connection` object, use the `apacheCassandra` function.

Properties

Cluster — Cassandra cluster name

string scalar

This property is read-only.

Cassandra cluster name, specified as a string scalar.

Example: "Test Cluster"

Data Types: string

HostAddresses — Host address

string scalar | string array

This property is read-only.

Host address for one node or host addresses for multiple nodes in the Cassandra cluster, specified as a string scalar for one node or string array for multiple nodes. These addresses include the addresses specified in the 'ContactPoints' name-value argument of the `apacheCassandra` function.

Example: "localhost/127.0.0.1"

Data Types: string

LocalDataCenter — Local data center name

string scalar

This property is read-only.

Local data center name, specified as a string scalar. The name describes the data center that the cluster declares as local for the connection instance. The name matches the data center of the

original contact point for the connection. When you set up the cluster, you define a data center for each node.

Example: "datacenter1"

Data Types: string

Keyspaces — Keyspaces

string scalar | string array

This property is read-only.

Keyspaces in the Cassandra database, specified as a string scalar or string array. A string scalar indicates the Cassandra database has one keyspace, and a string array indicates multiple keyspaces.

Example: ["employeedata" "system"]

Data Types: string

RequestTimeout — Request timeout

12 (default) | positive numeric scalar

This property is read-only.

Request timeout, specified as a positive numeric scalar. The request timeout indicates the number of seconds to wait for the database to return a CQL query before throwing an error.

Data Types: double

Object Functions

Cassandra Database Connection

close Close Apache Cassandra database connection

isopen Determine if Apache Cassandra database connection is open

Import Cassandra Database Data into MATLAB

columninfo Retrieve column information from Apache Cassandra database table

partitionRead Import data from partitions of Apache Cassandra database table

tablenames List names of database tables in Apache Cassandra database

Export MATLAB Data into Cassandra Database

upsert Insert or update data in Apache Cassandra database

Execute CQL Query

executecql Execute CQL query on Apache Cassandra database

Examples

Create Apache Cassandra Database Connection

Create a database connection to an Apache™ Cassandra® database using the Apache Cassandra database C++ interface. To create this connection, you must configure a Cassandra data source. For

more information, see the `databaseConnectionOptions` function. Using a local host address, create the database connection and display the keyspaces in the database.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password)
```

```
conn =
  connection with properties:
      Cluster: "Test Cluster"
      HostAddresses: "127.0.0.1"
      LocalDataCenter: "datacenter1"
      RequestTimeout: 20
      Keyspaces: [6×1 string]
```

`conn` is a connection object that contains these properties:

- Cassandra cluster name
- Host address
- Local data center name
- Keyspaces
- Request timeout

Display the keyspaces in the Cassandra database by accessing the `Keyspaces` property of the connection object.

```
conn.Keyspaces
```

```
ans = 6×1 string
      "employeeedata"
      "system"
      "system_auth"
      "system_distributed"
      "system_schema"
      "system_traces"
```

Close the Cassandra database connection.

```
close(conn)
```

Version History

Introduced in R2021a

See Also

Objects

CassandraConnectionOptions

Functions

databaseConnectionOptions | deleteDataSource

External Websites

Apache Cassandra

Apache Cassandra Documentation

apacheCassandra

Create Apache Cassandra database connection

Syntax

```
conn = apacheCassandra(datasource,username,password)
conn = apacheCassandra(username,password,Name,Value)
```

Description

`conn = apacheCassandra(datasource,username,password)` creates a Cassandra database connection using a data source name, user name, and password.

`conn = apacheCassandra(username,password,Name,Value)` specifies options using one or more name-value arguments. For example, 'PortNumber',9042 creates a Cassandra database connection using the port number 9042.

Examples

Create Apache Cassandra Database Connection

Create a database connection to an Apache™ Cassandra® database using the Apache Cassandra database C++ interface. To create this connection, you must configure a Cassandra data source. For more information, see the `databaseConnectionOptions` function. Using a local host address, create the database connection and display the keyspaces in the database.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password)
```

```
conn =
  connection with properties:
      Cluster: "Test Cluster"
      HostAddresses: "127.0.0.1"
      LocalDataCenter: "datacenter1"
      RequestTimeout: 20
      Keyspaces: [6×1 string]
```

`conn` is a connection object that contains these properties:

- Cassandra cluster name
- Host address
- Local data center name

- Keyspaces
- Request timeout

Display the keyspaces in the Cassandra database by accessing the Keyspaces property of the connection object.

```
conn.Keyspaces
```

```
ans = 6x1 string
      "employeeedata"
      "system"
      "system_auth"
      "system_distributed"
      "system_schema"
      "system_traces"
```

Close the Cassandra database connection.

```
close(conn)
```

Create Apache Cassandra Database Connection Using Additional Options

Create a database connection to an Apache™ Cassandra® database with additional options using the Apache Cassandra database C++ interface. Using an additional option for request timeout, create the database connection and display the keyspaces in the database.

Create a Cassandra database connection using a blank user name and password. Specify an additional option for a request timeout of 20 seconds.

```
username = "";
password = "";
conn = apacheCassandra(username,password, ...
    'RequestTimeout',20)
```

```
conn =
  connection with properties:
      Cluster: "Test Cluster"
      HostAddresses: "127.0.0.1"
      LocalDataCenter: "datacenter1"
      RequestTimeout: 20
      Keyspaces: [6x1 string]
```

conn is a connection object that contains these properties:

- Cassandra cluster name
- Host address
- Local data center name
- Keyspaces
- Request timeout

Display the keyspaces in the Cassandra database by accessing the `Keyspaces` property of the `connection` object.

```
conn.Keyspaces  
  
ans = 6x1 string  
    "employeeedata"  
    "system"  
    "system_auth"  
    "system_distributed"  
    "system_schema"  
    "system_traces"
```

Close the Cassandra database connection.

```
close(conn)
```

Input Arguments

datasource — Data source name

character vector | string scalar

Data source name, specified as a character vector or string scalar. Specify the name of an existing data source.

Example: "myDataSource"

Data Types: char | string

username — User name

character vector | string scalar

User name, specified as a character vector or string scalar. If the cluster requires authentication, use the `username` input argument for the user name.

Data Types: char | string

password — Password

character vector | string scalar

Password, specified as a character vector or string scalar. If the cluster requires authentication, use the `password` input argument for the password.

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `conn = apacheCassandra(username,password,'SSEnabled',true,'RequestTimeout',15)`

creates a Cassandra database connection with SSL encryption enabled and a request timeout of 15 seconds.

ContactPoints – Contact points

"localhost" (default) | character vector | string scalar | cell array of character vectors | string array

Contact points, specified as a character vector, string scalar, cell array of character vectors, or string array. Contact points are host addresses for one node or for multiple nodes in the Cassandra cluster. Specify a string scalar for one node. Or, specify a string array for multiple nodes.

You can specify a local host or the IP address of a different machine in the Cassandra cluster.

When you specify multiple nodes, if the connection to one host fails, then the `apacheCassandra` function connects to the other nodes in the cell array or string array until a connection succeeds. If a connection attempt fails for all specified nodes, the function displays an error message. If one or more nodes are not available, enter multiple nodes in the cell array or string array to increase the likelihood of a successful connection.

Data Types: `char` | `string` | `cell`

PortNumber – Port number

9042 (default) | positive numeric scalar

Port number for connection to the host, specified as a positive numeric scalar.

Data Types: `double`

SSLEnabled – SSL-enabled connection

false (default) | true

SSL-enabled connection, specified as the value `false` or `true`. Setting this argument to `true` creates an SSL-enabled connection to the Cassandra database.

Data Types: `logical`

LoginTimeout – Login timeout

5 (default) | positive numeric scalar

Login timeout, specified as a positive numeric scalar. The login timeout specifies the number of seconds that the C++ driver waits while trying to connect to the Cassandra database before throwing an error.

Data Types: `double`

RequestTimeout – Request timeout

12 (default) | positive numeric scalar

This property is read-only.

Request timeout, specified as a positive numeric scalar. The request timeout indicates the number of seconds the database waits to return a CQL query before throwing an error.

Data Types: `double`

Output Arguments

conn — Apache Cassandra database connection

`connection` object

Apache Cassandra database connection, returned as a `connection` object.

Version History

Introduced in R2021a

See Also

Objects

`CassandraConnectionOptions` | `connection`

Functions

`isopen` | `tablenames` | `columninfo` | `partitionRead` | `upsert` | `executecql` | `close`

External Websites

[Apache Cassandra](#)

[Apache Cassandra Documentation](#)

isopen

Package: database.apacheCassandra

Determine if Apache Cassandra database connection is open

Syntax

```
val = isopen(conn)
```

Description

`val = isopen(conn)` returns 1 if the Cassandra database connection is open and 0 if it is closed.

Examples

Verify Cassandra Connection

Using the Apache™ Cassandra® database C++ interface, create a Cassandra database connection and verify that the connection is open. Display the keyspaces in the Cassandra database, and then close the connection.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password);
```

Verify that the Cassandra database connection is open.

```
val = isopen(conn)
```

```
val = logical
    1
```

The database connection is open because the `isopen` function returns 1. Otherwise, the database connection is closed.

Display the keyspaces in the Cassandra database.

```
conn.Keyspaces
```

```
ans = 6x1 string
    "employeeedata"
    "system"
    "system_auth"
    "system_distributed"
    "system_schema"
```

```
"system_traces"
```

Close the Cassandra database connection.

```
close(conn)
```

Verify that the Cassandra database connection is closed.

```
val = isopen(conn)
```

```
val = logical  
    0
```

Input Arguments

conn — Apache Cassandra database connection

connection object

Apache Cassandra database connection, specified as a *connection* object.

Version History

Introduced in R2021a

See Also

Objects

CassandraConnectionOptions | *connection*

Functions

apacheCassandra | *tablename* | *columninfo* | *partitionRead* | *executecql* | *close*

Topics

“Explore and Import Data from Cassandra Database Table” on page 8-11

“Import Data from Cassandra Database Table Using CQL” on page 8-14

External Websites

Apache Cassandra Documentation

close

Package: database.apacheCassandra

Close Apache Cassandra database connection

Syntax

```
close(conn)
```

Description

`close(conn)` closes the Cassandra database connection.

Examples

Create Apache Cassandra Database Connection

Create a database connection to an Apache™ Cassandra® database using the Apache Cassandra database C++ interface. To create this connection, you must configure a Cassandra data source. For more information, see the `databaseConnectionOptions` function. Using a local host address, create the database connection and display the keyspaces in the database.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password.

```
datasource = "CassandraDataSource";  
username = "";  
password = "";  
conn = apacheCassandra(datasource,username,password)
```

```
conn =  
  connection with properties:  
    Cluster: "Test Cluster"  
    HostAddresses: "127.0.0.1"  
    LocalDataCenter: "datacenter1"  
    RequestTimeout: 20  
    Keyspaces: [6×1 string]
```

`conn` is a `connection` object that contains these properties:

- Cassandra cluster name
- Host address
- Local data center name
- Keyspaces
- Request timeout

Display the keyspaces in the Cassandra database by accessing the `Keyspaces` property of the connection object.

```
conn.Keyspaces
```

```
ans = 6x1 string
    "employeeedata"
    "system"
    "system_auth"
    "system_distributed"
    "system_schema"
    "system_traces"
```

Close the Cassandra database connection.

```
close(conn)
```

Input Arguments

conn — Apache Cassandra database connection

connection object

Apache Cassandra database connection, specified as a connection object.

Version History

Introduced in R2021a

See Also

Objects

`CassandraConnectionOptions` | `connection`

Functions

`apacheCassandra` | `isopen` | `columninfo` | `tablenames` | `partitionRead` | `executecql`

Topics

“Explore and Import Data from Cassandra Database Table” on page 8-11

“Import Data from Cassandra Database Table Using CQL” on page 8-14

External Websites

Apache Cassandra Documentation

tablenames

Package: database.apacheCassandra

List names of database tables in Apache Cassandra database

Syntax

```
t = tablenames(conn)
t = tablenames(conn, keyspace)
```

Description

`t = tablenames(conn)` returns a list that contains the names of Cassandra database tables and their corresponding keyspaces of the Cassandra database.

`t = tablenames(conn, keyspace)` returns a list that contains the names of Cassandra database tables in the specified keyspace of the Cassandra database.

Examples

Return Database Table Names in Cassandra Database

Using a Cassandra® database connection and the Apache™ Cassandra database C++ interface, return the names of all database tables in the Cassandra database.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource, username, password);
```

Return the names of all database tables in the Cassandra database using the Cassandra database connection. `t` is a table that contains the names of all Cassandra database tables and their corresponding keyspaces.

```
t = tablenames(conn);
```

Display the first few rows of the returned data.

```
head(t)
```

```
ans=8x2 table
  Keyspace      Table
-----
"employeeedata" "employees_by_id"
"employeeedata" "employees_by_job"
"employeeedata" "employees_by_name"
```

```
"system"      "IndexInfo"  
"system"      "available_ranges"  
"system"      "batches"  
"system"      "batchlog"  
"system"      "built_views"
```

The `Keyspace` variable indicates the keyspace. The `Table` variable indicates the name of the Cassandra database table in the corresponding keyspace.

Close the Cassandra database connection.

```
close(conn)
```

Return Database Table Names in Cassandra Database Keyspace

Using an Apache™ Cassandra® database connection and the Apache Cassandra database C++ interface, return the names of all database tables in a specific keyspace of the Cassandra database—in this case, the `employeedata` keyspace.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a `connection` object.

```
datasource = "CassandraDataSource";  
username = "";  
password = "";  
conn = apacheCassandra(datasource,username,password);
```

Return and display all database tables in the `employeedata` keyspace of the Cassandra database by using the Cassandra database connection. `t` is a string array that contains the names of all database tables in the `employeedata` keyspace.

```
keyspace = "employeedata";  
t = tablename(conn,keyspace)
```

```
t = 3x1 string  
    "employees_by_id"  
    "employees_by_job"  
    "employees_by_name"
```

Close the Cassandra database connection.

```
close(conn)
```

Input Arguments

conn — Apache Cassandra database connection

`connection` object

Apache Cassandra database connection, specified as a `connection` object.

keyspace — Keyspace

character vector | string scalar

Keyspace, specified as a character vector or string scalar. If you do not know the keyspace, then access the `Keyspaces` property of the `connection` object using dot notation to view the keyspaces in the Cassandra database.

Example: "employeedata"

Data Types: char | string

Output Arguments**t — Database table names**

string array | table

Database table names in the Cassandra database, specified as a string array or table. If you specify a keyspace in the `keyspace` input argument, the `tablenames` function returns a string array that contains all the database table names in the specified keyspace of the Cassandra database. If you do not specify a keyspace, the `tablenames` function returns a table with the `Keyspace` and `Table` variables. The `Keyspace` variable is a string array that contains all keyspaces in the Cassandra database. The `Table` variable is a string array that contains the names of all database tables in the Cassandra database for their corresponding keyspaces.

Version History

Introduced in R2021a

See Also**Objects**

CassandraConnectionOptions | connection

Functions

apacheCassandra | columninfo | partitionRead | upsert | executecql | close

Topics

"Explore and Import Data from Cassandra Database Table" on page 8-11

External Websites

Apache Cassandra Documentation

columninfo

Package: database.apacheCassandra

Retrieve column information from Apache Cassandra database table

Syntax

```
cols = columninfo(conn, keyspace, tablename)
[cols, keyValues] = columninfo(conn, keyspace, tablename)
```

Description

`cols = columninfo(conn, keyspace, tablename)` returns column information from a specified Cassandra database table in a specified keyspace using the Cassandra database connection.

`[cols, keyValues] = columninfo(conn, keyspace, tablename)` also returns the key values for each partition in the Cassandra database table.

Examples

Return Column Information from Cassandra Database Table

Using an Apache™ Cassandra® database connection and the Apache Cassandra database C++ interface, return column information for a Cassandra database table. Specify the keyspace and the name of the table. In this case, the Cassandra database has the `employeedata` keyspace, which contains the `employees_by_job` database table.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource, username, password);
```

Return column information for the `employees_by_job` database table in the `employeedata` keyspace.

```
keyspace = "employeedata";
tablename = "employees_by_job";
cols = columninfo(conn, keyspace, tablename);
```

Display the first few rows of column information.

```
head(cols)
```

```
ans=8x4 table
      Name           DataType      PartitionKey      ClusteringColumn
```

```

"job_id"           "text"           true             ""
"hire_date"       "date"           false            "DESC"
"employee_id"     "int"            false            "ASC"
"commission_pct"  "double"         false            ""
"department_id"   "int"            false            ""
"email"           "text"           false            ""
"first_name"      "text"           false            ""
"last_name"       "text"           false            ""

```

`cols` is a table with these variables:

- `Name` — Cassandra database column name
- `DataType` — Cassandra Query Language (CQL) data type of the Cassandra database column
- `PartitionKey` — Partition key indicator
- `ClusteringColumn` — Clustering column indicator

Close the Cassandra database connection.

```
close(conn)
```

Return Partition Key Values from Cassandra Database Table

Using an Apache™ Cassandra® database connection and the Apache Cassandra database C++ interface, return partition key values for a Cassandra database table. Specify the keyspace and name of the table. In this case, the Cassandra database has the `employeedata` keyspace, which contains the `employees_by_job` database table.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```

datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password);

```

Return column information for the `employees_by_job` database table in the `employeedata` keyspace.

```

keyspace = "employeedata";
tablename = "employees_by_job";
[cols,keyValues] = columninfo(conn,keyspace,tablename);

```

`keyValues` is a table that contains a variable for each partition key. The rows are partition key values.

Display the first few partition key values of the Cassandra database table.

```
head(keyValues)
```

```

ans=8x1 table
  job_id

```

```
"ST_CLERK"  
"SA_MAN"  
"HR_REP"  
"IT_PROG"  
"FI_MGR"  
"PR_REP"  
"PU_MAN"  
"AD_PRES"
```

`job_id` is the only partition key in the `employees_by_job` database table. Each row is a partition key value that is a unique partition in `employees_by_job`.

Use the partition key values with the `partitionRead` function to import data from a Cassandra database table.

Close the Cassandra database connection.

```
close(conn)
```

Input Arguments

conn — Apache Cassandra database connection

connection object

Apache Cassandra database connection, specified as a `connection` object.

keyspace — Keyspace

character vector | string scalar

Keyspace, specified as a character vector or string scalar. If you do not know the keyspace, then access the `Keyspaces` property of the `connection` object using dot notation to view the keyspaces in the Cassandra database.

Example: `"employeedata"`

Data Types: `char` | `string`

tablename — Cassandra database table name

character vector | string scalar

Cassandra database table name, specified as a character vector or string scalar. If you do not know the name of the table, then use the `tablenames` function to find it.

Example: `"employees_by_job"`

Data Types: `char` | `string`

Output Arguments

cols — Column information

table

Column information from a Cassandra database table, returned as a MATLAB table containing these variables.

Variable Name	Variable Description	Variable Data Type
Name	Cassandra database column name	string
DataType	CQL data type of the Cassandra database column	string
PartitionKey	Whether the Cassandra database table column is a partition key (true indicates a partition key)	logical
ClusteringColumn	Whether the Cassandra database table column is a clustering column ("ASC" indicates ascending order, "DESC" indicates descending order, and "" indicates that the column is not a clustering column)	string

If the data type of a column in a Cassandra database table is a collection (for example, a list, map, and so on), then the value of the `DataType` variable contains angle brackets (<>). These angle brackets surround the data type of the items in the collection. The value for user-defined types (UDTs) contains the type name. For example, if the UDT is `person`, then the value of the `DataType` variable is `person` in the MATLAB table. For details about valid CQL data types, see [CQL Data Types](#).

keyValues — Partition key values

table

Partition key values, returned as a table. The MATLAB table contains one variable for each partition key in the Cassandra database table. Each row in the MATLAB table represents a unique partition in the Cassandra database table.

You can use the partition key values with the `partitionRead` function to import data from a Cassandra database table.

For details about the MATLAB data types of the partition key values, see “Convert CQL Data Types to MATLAB Data Types Using Apache Cassandra Database C++ Interface” on page 8-2.

Version History

Introduced in R2021a

See Also

Objects

`CassandraConnectionOptions` | `connection`

Functions

`apacheCassandra` | `tablename` | `partitionRead` | `upsert` | `executecql` | `close`

Topics

“Explore and Import Data from Cassandra Database Table” on page 8-11

External Websites

[Apache Cassandra Documentation](#)
[CQL Data Types](#)

upsert

Package: database.apacheCassandra

Insert or update data in Apache Cassandra database

Syntax

```
upsert(conn, keyspace, tablename, data)
upsert(conn, keyspace, tablename, data, Name, Value)
```

Description

`upsert(conn, keyspace, tablename, data)` exports data from MATLAB by inserting or updating it in an Apache Cassandra database table.

`upsert(conn, keyspace, tablename, data, Name, Value)` specifies options using one or more name-value arguments. For example, 'ConsistencyLevel', "TWO" sets the consistency level to specify that two nodes must respond for the CQL query to execute.

Examples

Insert MATLAB Data into Cassandra Database

Insert data from MATLAB® into an Apache™ Cassandra® database using the Apache Cassandra database C++ interface, and display the data by using a Cassandra database connection.

The Cassandra database includes the `employees_by_job` database table, which contains employee data and the `job_id` partition key.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource, username, password);
```

Return the names of the Cassandra database tables in the `employeedata` keyspace. `t` is a string array that contains the names of these tables.

```
keyspace = "employeedata";
t = tablename(conn, keyspace)
```

```
t = 3x1 string
    "employees_by_id"
    "employees_by_job"
    "employees_by_name"
```

Import employee data into MATLAB from the employees_by_job table by using the Cassandra database connection.

```
keyspace = "employeeedata";
tablename = "employees_by_job";
results = partitionRead(conn, keyspace, tablename);
```

Display the last few rows of the imported employee data.

```
tail(results)
```

```
ans=8x13 table
```

job_id	hire_date	employee_id	commission_pct	department_id	email
"SH_CLERK"	27-Jan-2004	184	NaN	50	"NSARCHAN"
"MK_REP"	17-Aug-2005	202	NaN	20	"PFAY"
"PU_CLERK"	10-Aug-2007	119	NaN	30	"KCOLMENA"
"PU_CLERK"	15-Nov-2006	118	NaN	30	"GHIMURO"
"PU_CLERK"	24-Dec-2005	116	NaN	30	"SBAIDA"
"PU_CLERK"	24-Jul-2005	117	NaN	30	"STOBIAS"
"PU_CLERK"	18-May-2003	115	NaN	30	"AKHOO"
"AC_ACCOUNT"	07-Jun-2002	206	NaN	110	"WGIETZ"

results is a table that contains these variables:

- job_id — Job identifier
- hire_date — Hire date
- employee_id — Employee identifier
- commission_pct — Commission percentage
- department_id — Department identifier
- email — Email address
- first_name — First name
- last_name — Last name
- manager_id — Manager identifier
- office — Office location (table that contains two variables for the building and room)
- performance_ratings — Performance ratings
- phone_number — Phone number
- salary — Salary

Display the CQL data types of the columns in the employees_by_job database table.

```
cols = columninfo(conn, keyspace, tablename);
cols(:,1:2)
```

```
ans=13x2 table
```

Name	DataType
"job_id"	"text"
"hire_date"	"date"

```

"employee_id"      "int"
"commission_pct"   "double"
"department_id"    "int"
"email"            "text"
"first_name"       "text"
"last_name"        "text"
"manager_id"       "int"
"office"           "office"
"performance_ratings" "list<int>"
"phone_number"     "text"
"salary"           "int"

```

Create a table of data representing one employee to insert into the Cassandra database. Specify the names of the variables. Create a table for the office information. Then, create a table with the employee information that contains the nested table of office information. Set the names of the variables.

```

varnames = ["job_id" "hire_date" "employee_id" ...
            "commission_pct" "department_id" "email" "first_name" ...
            "last_name" "manager_id" "office" "performance_ratings" ...
            "phone_number" "salary"];
office = table("South",160, ...
              'VariableNames',["building" "room"]);
data = table("IT_ADMIN",datetime('today'),301,0.25,30,"SMITH123", ...
            "Alex","Smith",114,office,{[4 5]},"515.123.2345",3000);
data.Properties.VariableNames = varnames;

```

Insert the employee information into the Cassandra database.

```
upsert(conn, keyspace, tablename, data)
```

Display the inserted data by importing it into MATLAB using the partition key IT_ADMIN. The employees_by_job table contains a new row.

```
keyValue = "IT_ADMIN";
results = partitionRead(conn, keyspace, tablename, keyValue)
```

```

results=1x13 table
      job_id      hire_date      employee_id      commission_pct      department_id      email
      _____      _____      _____      _____      _____      _____
      "IT_ADMIN"      06-Oct-2020      301      0.25      30      "SMITH123"

```

Close the Cassandra database connection.

```
close(conn)
```

Update Data in Cassandra Database Using Consistency Level

Using the Apache™ Cassandra® database C++ interface, update data in an Apache Cassandra database with MATLAB® data. Display the updated data by using a Cassandra database connection. Specify a consistency level for the write operation.

The Cassandra database includes the `employees_by_job` database table, which contains employee data and the `job_id` partition key.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password);
```

Return the names of the Cassandra database tables in the `employeedata` keyspace. `t` is a string array that contains the names of these tables.

```
keyspace = "employeedata";
t = tablenames(conn,keyspace)
```

```
t = 3x1 string
    "employees_by_id"
    "employees_by_job"
    "employees_by_name"
```

Import employee data into MATLAB from the `employees_by_job` table by using the Cassandra database connection.

```
keyspace = "employeedata";
tablename = "employees_by_job";
results = partitionRead(conn,keyspace,tablename);
```

Display the last few rows of the imported employee data.

```
tail(results)
```

```
ans=8x13 table
    job_id      hire_date      employee_id      commission_pct      department_id      email
    _____    _____    _____    _____    _____    _____
    "SH_CLERK"    27-Jan-2004      184              NaN                50              "NSARCHAN"
    "MK_REP"      17-Aug-2005      202              NaN                20              "PFAY"
    "PU_CLERK"    10-Aug-2007      119              NaN                30              "KCOLMENA"
    "PU_CLERK"    15-Nov-2006      118              NaN                30              "GHIMURO"
    "PU_CLERK"    24-Dec-2005      116              NaN                30              "SBAIDA"
    "PU_CLERK"    24-Jul-2005      117              NaN                30              "STOBIAS"
    "PU_CLERK"    18-May-2003      115              NaN                30              "AKHOO"
    "AC_ACCOUNT"  07-Jun-2002      206              NaN                110             "WGIETZ"
```

`results` is a table that contains these variables:

- `job_id` — Job identifier
- `hire_date` — Hire date
- `employee_id` — Employee identifier
- `commission_pct` — Commission percentage

- `department_id` — Department identifier
- `email` — Email address
- `first_name` — First name
- `last_name` — Last name
- `manager_id` — Manager identifier
- `office` — Office location (table that contains two variables for the building and room)
- `performance_ratings` — Performance ratings
- `phone_number` — Phone number
- `salary` — Salary

Display the CQL data types of the columns in the `employees_by_job` database table.

```
cols = columninfo(conn, keyspace, tablename);
cols(:, 1:2)
```

```
ans=13x2 table
      Name      DataType
-----
"job_id"      "text"
"hire_date"   "date"
"employee_id" "int"
"commission_pct" "double"
"department_id" "int"
"email"       "text"
"first_name"  "text"
"last_name"   "text"
"manager_id"  "int"
"office"      "office"
"performance_ratings" "list<int>"
"phone_number" "text"
"salary"      "int"
```

Import the data to update by using the `partitionRead` function with the partition key value `MK_REP`. The data is for an employee who is a marketing representative.

```
keyValue = "MK_REP";
data = partitionRead(conn, keyspace, tablename, keyValue)
```

```
data=1x13 table
  job_id  hire_date  employee_id  commission_pct  department_id  email  first_
-----
"MK_REP" 17-Aug-2005      202          NaN           20          "PFAY"  "Pa
```

Update the commission percentage to 0.25 for the marketing representative. Also, specify the consistency level `"ONE"` to ensure that one replica node commits the write operation.

```
data.commission_pct = 0.25;
level = "ONE";
upsert(conn, keyspace, tablename, data, 'ConsistencyLevel', level)
```

Display the updated data by importing it into MATLAB using the partition key value MK_REP. The updated commission percentage for the marketing representative is 0.25.

```
keyValue = "MK_REP";
results = partitionRead(conn, keyspace, tablename, keyValue)
```

```
results=1x13 table
      job_id      hire_date      employee_id      commission_pct      department_id      email      first_
      _____      _____      _____      _____      _____      _____      _____
      "MK_REP"      17-Aug-2005      202      0.25      20      "PFAY"      "Pa
```

Close the Cassandra database connection.

```
close(conn)
```

Input Arguments

conn — Apache Cassandra database connection

connection object

Apache Cassandra database connection, specified as a `connection` object.

keyspace — Keyspace

character vector | string scalar

Keyspace, specified as a character vector or string scalar. If you do not know the keyspace, then access the `Keyspaces` property of the connection object using dot notation to view the keyspaces in the Cassandra database.

Example: "employeedata"

Data Types: `char` | `string`

tablename — Cassandra database table name

character vector | string scalar

Cassandra database table name, specified as a character vector or string scalar. If you do not know the name of the table, then use the `tablenames` function to find it.

Example: "employees_by_job"

Data Types: `char` | `string`

data — Data to insert or update

table

Data to insert or update in a Cassandra database, specified as a table. You must specify the primary keys of the Cassandra database table, but you can ignore other Cassandra columns. The names of the variables in the table must match the names of the Cassandra columns in the database table, without case sensitivity. The data types of the variables in the table must be compatible with the CQL data types of the Cassandra columns. For details, see "Convert CQL Data Types to MATLAB Data Types Using Apache Cassandra Database C++ Interface" on page 8-2.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example:

```
upsert(conn, keyspace, tablename, data, 'ConsistencyLevel', "ONE", 'RequestTimeout', 15)
```

exports data by receiving a write response from one node, and the database must wait 15 seconds to perform the write operation before throwing an error.

ConsistencyLevel – Consistency level

"ONE" (default) | character vector | string scalar

Consistency level, specified as one of these values.

Consistency Level Value	Write Operation
"ALL"	Commit on all replica nodes.
"EACH_QUORUM"	Commit on a majority of replica nodes in each data center.
"QUORUM"	Commit on most replica nodes.
"LOCAL_QUORUM"	Commit on most replica nodes in the local data center.
"ONE" (default)	Commit on one replica node.
"TWO"	Commit on two replica nodes.
"THREE"	Commit on three replica nodes.
"LOCAL_ONE"	Commit on one replica node in the local data center.
"ANY"	Commit on at least one replica node.

You can specify the value of the consistency level as a character vector or string scalar.

For details about consistency levels, see [Configuring Data Consistency](#).

Data Types: `char` | `string`

RequestTimeout – Request timeout

12 (default) | positive numeric scalar

This property is read-only.

Request timeout, specified as a positive numeric scalar. The request timeout indicates the number of seconds the database waits to return a CQL query before throwing an error.

Data Types: `double`

Version History

Introduced in R2021a

See Also

Objects

connection

Functions

apacheCassandra | tablenamees | columninfo | partitionRead | executecql | close

Topics

“Explore and Import Data from Cassandra Database Table” on page 8-11

“Export MATLAB Data into Cassandra Database” on page 8-16

“Convert CQL Data Types to MATLAB Data Types Using Apache Cassandra Database C++ Interface” on page 8-2

External Websites

Apache Cassandra Documentation

CQL Data Types

Configuring Data Consistency

partitionRead

Package: database.apacheCassandra

Import data from partitions of Apache Cassandra database table

Syntax

```
results = partitionRead(conn, keyspace, tablename)
results = partitionRead(conn, keyspace, tablename, keyValue1...keyValueN)
results = partitionRead( ____, Name, Value)
```

Description

`results = partitionRead(conn, keyspace, tablename)` returns imported data by reading all Cassandra database columns from all partitions of a Cassandra database table. The `partitionRead` function imports data from a Cassandra database into MATLAB without using a Cassandra Query Language (CQL) query.

`results = partitionRead(conn, keyspace, tablename, keyValue1...keyValueN)` returns imported data by reading all Cassandra columns from one or more partitions specified by the partition key values.

`results = partitionRead(____, Name, Value)` specifies options using one or more name-value arguments in addition to any of the previous input argument combinations. For example, 'ConsistencyLevel', "TWO" sets the consistency level to specify that two nodes must respond for the CQL query to execute.

Examples

Import Data from Cassandra Database Table

Using the Apache™ Cassandra® database C++ interface, create a Cassandra database connection and import data from a Cassandra database table into MATLAB®. The Cassandra database contains a database table with employee data.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource, username, password);
```

Import employee data into MATLAB from the `employeedata` keyspace and the `employees_by_job` database table by using the Cassandra database connection.

```
keyspace = "employeedata";
tablename = "employees_by_job";
results = partitionRead(conn, keyspace, tablename);
```

Display the first few rows of the returned employee data.

```
head(results)
```

```
ans=8x13 table
      job_id      hire_date      employee_id      commission_pct      department_id      email
-----
"ST_CLERK"      08-Mar-2008           128              NaN              50              "SMARKLE"
"ST_CLERK"      06-Feb-2008           136              NaN              50              "HPHILTAN"
"ST_CLERK"      12-Dec-2007           135              NaN              50              "KGEE"
"ST_CLERK"      10-Apr-2007           132              NaN              50              "TJOLSON"
"ST_CLERK"      14-Jan-2007           127              NaN              50              "JLANDRY"
"ST_CLERK"      28-Sep-2006           126              NaN              50              "IMIKKILI"
"ST_CLERK"      26-Aug-2006           134              NaN              50              "MROGERS"
"ST_CLERK"      09-Jul-2006           144              NaN              50              "PVARGAS"
```

`results` is a table that contains these variables:

- `job_id` — Job identifier
- `hire_date` — Hire date
- `employee_id` — Employee identifier
- `commission_pct` — Commission percentage
- `department_id` — Department identifier
- `email` — Email address
- `first_name` — First name
- `last_name` — Last name
- `manager_id` — Manager identifier
- `office` — Office location (table that contains two variables for the building and room)
- `performance_ratings` — Performance ratings
- `phone_number` — Phone number
- `salary` — Salary

Close the Cassandra database connection.

```
close(conn)
```

Import Data Using Multiple Partition Key Values

Using the Apache™ Cassandra® database C++ interface, create a Cassandra® database connection and import data from a Cassandra database table into MATLAB®. Use the values of two partition keys in the database table to import data. The Cassandra database contains a database table with employee data.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```

datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password);

```

Import employee data into MATLAB from the `employeedata` keyspace and the `employees_by_name` database table by using the Cassandra database connection. This database table has the `first_name` and `last_name` partition keys. Specify the first and last names of two employees as values of the partition keys to import data for those two employees.

```

keyspace = "employeedata";
tablename = "employees_by_name";
keyValue1 = ["Christopher","Alexander"];
keyValue2 = ["Olsen","Hunold"];
results = partitionRead(conn,keyspace,tablename,keyValue1,keyValue2);

```

Display the returned employee data for the two employees.

```
results
```

```
results=2x13 table
   first_name   last_name   hire_date   employee_id   commission_pct   department_id
   _____   _____   _____   _____   _____   _____
   "Alexander"   "Hunold"   03-Jan-2006   103           NaN           60
   "Christopher" "Olsen"   30-Mar-2006   153           0.2           80

```

`results` is a table that contains these variables:

- `first_name` — First name
- `last_name` — Last name
- `hire_date` — Hire date
- `employee_id` — Employee identifier
- `commission_pct` — Commission percentage
- `department_id` — Department identifier
- `email` — Email address
- `job_id` — Job identifier
- `manager_id` — Manager identifier
- `office` — Office location (table that contains two variables for the building and room)
- `performance_ratings` — Performance ratings
- `phone_number` — Phone number
- `salary` — Salary

Close the Cassandra database connection.

```
close(conn)
```

Import Data Using Consistency Level

Using the Apache™ Cassandra® database C++ interface, create a Cassandra database connection and import data from a Cassandra database table into MATLAB®. Use the value of the partition key in the database table to import data. Specify a consistency level for returning results. The Cassandra database contains a database table with employee data.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password);
```

Import employee data into MATLAB from the `employeedata` keyspace and the `employees_by_job` database table by using the Cassandra database connection. This database table has the `job_id` partition key. Specify the `IT_PROG` value of the partition key to import all data for only those employees who are programmers. Also, specify the consistency level as one node.

```
keyspace = "employeedata";
tablename = "employees_by_job";
keyValue = "IT_PROG";
level = "ONE";
results = partitionRead(conn,keyspace,tablename,keyValue, ...
    'ConsistencyLevel',level);
```

One replica node responds with the returned data.

Display the returned employee data.

```
results
```

```
results=5x13 table
   job_id      hire_date      employee_id      commission_pct      department_id      email
   _____      _____      _____      _____      _____      _____
   "IT_PROG"      21-May-2007      104              NaN                60                "BERNST"
   "IT_PROG"      07-Feb-2007      107              NaN                60                "DLORENTZ"
   "IT_PROG"      05-Feb-2006      106              NaN                60                "VPATABAL"
   "IT_PROG"      03-Jan-2006      103              NaN                60                "AHUNOLD"
   "IT_PROG"      25-Jun-2005      105              NaN                60                "DAUSTIN"
```

`results` is a table that contains these variables:

- `job_id` — Job identifier
- `hire_date` — Hire date
- `employee_id` — Employee identifier
- `commission_pct` — Commission percentage
- `department_id` — Department identifier
- `email` — Email address

- `first_name` — First name
- `last_name` — Last name
- `manager_id` — Manager identifier
- `office` — Office location (table that contains two variables for the building and room)
- `performance_ratings` — Performance ratings
- `phone_number` — Phone number
- `salary` — Salary

Close the Cassandra database connection.

```
close(conn)
```

Input Arguments

conn — Apache Cassandra database connection

connection object

Apache Cassandra database connection, specified as a connection object.

keyspace — Keyspace

character vector | string scalar

Keyspace, specified as a character vector or string scalar. If you do not know the keyspace, then access the `Keyspaces` property of the `connection` object using dot notation to view the keyspaces in the Cassandra database.

Example: "employeeedata"

Data Types: char | string

tablename — Cassandra database table name

character vector | string scalar

Cassandra database table name, specified as a character vector or string scalar. If you do not know the name of the table, then use the `tablenames` function to find it.

Example: "employees_by_job"

Data Types: char | string

keyValue1...keyValueN — Partition key values

numeric scalar | numeric array | character vector | cell array of character vectors | ...

Partition key values, specified as one of these data types:

- numeric scalar
- numeric array
- character vector
- cell array of character vectors
- string scalar
- string array

- `logical`
- `logical` array
- `datetime` array
- `duration` array

If you do not specify the `keyValue1...keyValueN` input argument, then the `partitionRead` function imports data from all partitions of the Cassandra database table (same as the CQL query `SELECT * FROM tablename`).

Specify one key value for each partition key of the Cassandra database table. The maximum number of partition key values that you can specify is the number of primary keys, which includes the partition keys and clustering columns in the Cassandra database.

If you specify a scalar value, then the CQL query equivalent is an `=` clause in the CQL `WHERE` clause. If you specify an array of values, then the CQL query equivalent is an `IN` clause in the CQL `WHERE` clause.

If all partition key values are scalar values, then the `partitionRead` function imports data from one partition. If some partition key values are arrays, then the `partitionRead` function imports data by searching multiple partitions that correspond to all possible key combinations.

The following table describes supported Cassandra partition keys.

Supported Cassandra Partition Key	MATLAB Valid Data Types for One Partition	MATLAB Valid Data Types for Multiple Partitions
<code>ascii</code>	character vector or string scalar	cell array of character vectors or string array
<code>bigint</code>	numeric scalar or <code>logical</code> scalar	numeric array or <code>logical</code> array
<code>blob</code>	numeric array	cell array of numeric arrays
<code>boolean</code>	numeric scalar or <code>logical</code> scalar	numeric array or <code>logical</code> array
<code>date</code>	<code>datetime</code> array, string scalar, or character vector	<code>datetime</code> array, string array, or cell array of character vectors
<code>decimal</code>	numeric scalar, <code>logical</code> scalar, or string scalar	numeric array, <code>logical</code> array, or string array
<code>double</code>	numeric scalar or <code>logical</code> scalar	numeric array or <code>logical</code> array
<code>float</code>	numeric scalar or <code>logical</code> scalar	numeric array or <code>logical</code> array
<code>inet</code>	character vector or string scalar	cell array of character vectors or string array
<code>int</code>	numeric scalar or <code>logical</code> scalar	numeric array or <code>logical</code> array
<code>smallint</code>	numeric scalar or <code>logical</code> scalar	numeric array or <code>logical</code> array
<code>text</code>	character vector or string scalar	cell array of character vectors or string array
<code>time</code>	<code>duration</code> array, string scalar, or character vector	<code>duration</code> array, string array, or cell array of character vectors
<code>timestamp</code>	<code>datetime</code> array, string scalar, or character vector	<code>datetime</code> array, string array, or cell array of character vectors

Supported Cassandra Partition Key	MATLAB Valid Data Types for One Partition	MATLAB Valid Data Types for Multiple Partitions
timeuuid	character vector or string scalar	cell array of character vectors or string array
tinyint	numeric scalar or logical scalar	numeric array or logical array
uuid	character vector or string scalar	cell array of character vectors or string array
varchar	character vector or string scalar	cell array of character vectors or string array
varint	numeric scalar, logical scalar, or string	numeric array, logical array, or string array

These Cassandra partition keys are not supported:

- counter
- list
- map
- set
- tuple
- user-defined types (UDTs)

Example: ["MA", "CT"]

Example: 1,2, 'DataProvider1', 'AmbientTemp'

Data Types: double | logical | char | string | struct | cell | datetime | duration

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: results =

partitionRead(conn, keyspace, tablename, 'ConsistencyLevel', "ONE", 'RequestTimeout', 15) returns imported data by receiving a read response from one node, and the database must wait 15 seconds to perform the read operation before throwing an error.

ConsistencyLevel – Consistency level

"ONE" (default) | character vector | string scalar

Consistency level, specified as one of these values.

Consistency Level Value	Consistency Level Description
"ALL"	Return query results when all replica nodes respond.
"QUORUM"	Return query results when most replica nodes respond.

Consistency Level Value	Consistency Level Description
"LOCAL_QUORUM"	Return query results when most replica nodes in the local data center respond.
"ONE" (default)	Return query results when one replica node responds.
"TWO"	Return query results when two replica nodes respond.
"THREE"	Return query results when three replica nodes respond.
"LOCAL_ONE"	Return query results when one replica node in the local data center responds.
"SERIAL"	Return query results for current (and possibly uncommitted) data for replica nodes in any data center.
"LOCAL_SERIAL"	Return query results for current (and possibly uncommitted) data for replica nodes in the local data center.

You can specify the value of the consistency level as a character vector or string scalar.

For details about consistency levels, see [Configuring Data Consistency](#).

Data Types: `char` | `string`

RequestTimeout — Request timeout

12 (default) | positive numeric scalar

This property is read-only.

Request timeout, specified as a positive numeric scalar. The request timeout indicates the number of seconds the database waits to return a CQL query before throwing an error.

Data Types: `double`

Output Arguments

results — Imported data results

table

Imported data results, returned as a table. The table contains imported data from the partitions that correspond to the `keyValue1...keyValueN` input argument. Each Cassandra database column from the partitions becomes a variable in the table. The variable names match the names of the Cassandra database columns in the specified partitions.

The data types of the variables in the table depend on the Cassandra data types. For details about how CQL data types convert to MATLAB data types, see “Convert CQL Data Types to MATLAB Data Types Using Apache Cassandra Database C++ Interface” on page 8-2.

Version History

Introduced in R2021a

See Also

Objects

`connection`

Functions

apacheCassandra | tablenamees | columninfo | upsert | executecql | close

Topics

“Explore and Import Data from Cassandra Database Table” on page 8-11

“Export MATLAB Data into Cassandra Database” on page 8-16

“Convert CQL Data Types to MATLAB Data Types Using Apache Cassandra Database C++ Interface” on page 8-2

“Fill Values for Missing Data from Database Using Apache Cassandra Database C++ Interface” on page 8-9

External Websites

Apache Cassandra Documentation

CQL Reference Documentation

CQL Data Types

Configuring Data Consistency

executecql

Package: database.apacheCassandra

Execute CQL query on Apache Cassandra database

Syntax

```
results = executecql(conn,query)
results = executecql(conn,query,Name,Value)
```

Description

`results = executecql(conn,query)` returns the results of executing a Cassandra Query Language (CQL) query on the Cassandra database using the Cassandra database connection. The `executecql` function imports the query results into MATLAB.

`results = executecql(conn,query,Name,Value)` specifies options using one or more name-value arguments. For example, 'ConsistencyLevel', "TWO" sets the consistency level to specify that two nodes must respond for the CQL query to execute.

Examples

Execute CQL Query

Using an Apache™ Cassandra® database C++ interface, create a Cassandra® database connection and execute a CQL query to import data from a Cassandra database table into MATLAB®. In this case, the Cassandra database contains the `employees_by_job` database table with employee data in the `employeedata` keyspace.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password);
```

Write a CQL query that selects all employees who are programmers or shop clerks hired before April 30, 2006, and retrieves their job identifiers, hire dates, and email addresses. `job_id` is the partition key of the `employees_by_job` database table, and `hire_date` is a clustering column.

```
query = strcat("SELECT job_id,hire_date,email ", ...
    "FROM employeedata.employees_by_job ", ...
    "WHERE job_id IN ('IT_PROG','SH_CLERK') ", ...
    "AND hire_date < '2006-04-30';");
```

Execute the CQL query and display the first few rows of results.

```
results = executecql(conn,query);
head(results)
```

```
ans=8x3 table
   job_id      hire_date      email
   _____  _____  _____
  "IT_PROG"    05-Feb-2006    "VPATABAL"
  "IT_PROG"    03-Jan-2006    "AHUNOLD"
  "IT_PROG"    25-Jun-2005    "DAUSTIN"
  "SH_CLERK"   24-Apr-2006    "AWALSH"
  "SH_CLERK"   23-Feb-2006    "JFLEAUR"
  "SH_CLERK"   24-Jan-2006    "WTAYLOR"
  "SH_CLERK"   13-Aug-2005    "JDILLY"
  "SH_CLERK"   14-Jun-2005    "KCHUNG"
```

`results` is a table with the `job_id`, `hire_date`, and `email` variables. The `hire_date` variable is a datetime array and the `job_id` and `email` variables are string arrays.

Close the Cassandra database connection.

```
close(conn)
```

Execute CQL Query Using Consistency Level

Using the Apache™ Cassandra® database C++ interface, create a Cassandra database connection and execute a CQL query to import data from a Cassandra database table into MATLAB®. Specify a consistency level for returning query results. In this case, the Cassandra database contains the `employees_by_job` database table with employee data in the `employeedata` keyspace.

Create a Cassandra database connection using the configured data source `CassandraDataSource` and a blank user name and password. The `apacheCassandra` function returns `conn` as a connection object.

```
datasource = "CassandraDataSource";
username = "";
password = "";
conn = apacheCassandra(datasource,username,password);
```

Write a CQL query that selects all employees who are programmers and retrieves their hire dates and email addresses. `job_id` is the partition key of the `employees_by_job` database table. Limit the returned data to three rows.

```
query = strcat("SELECT hire_date,email ", ...
  "FROM employeedata.employees_by_job ", ...
  "WHERE job_id = 'IT_PROG'", ...
  "LIMIT 3;");
```

Execute the CQL query with the consistency level set to one node and display the results.

```
level = "ONE";
results = executecql(conn,query,'ConsistencyLevel',level)
```

```
results=3x2 table
   hire_date      email
```

21-May-2007	"BERNST"
07-Feb-2007	"DLORENTZ"
05-Feb-2006	"VPATABAL"

In this case, only one replica node responds with returned data. `results` is a table with the `hire_date` and `email` variables. The `hire_date` variable is a datetime array and the `email` variable is a string array.

Close the Cassandra database connection.

```
close(conn)
```

Input Arguments

conn — Apache Cassandra database connection

connection object

Apache Cassandra database connection, specified as a connection object.

query — CQL query

character vector | string scalar

CQL query, specified as a character vector or string scalar. For details about CQL, see the Apache Software Foundation's CQL Reference Documentation.

Example: "SELECT * FROM dev.maps"

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `results =`

`executecql(conn,query,'ConsistencyLevel','TWO','RequestTimeout',15)` specifies to return query results when two nodes respond for the CQL query execution, and the database must wait 15 seconds to return the query before throwing an error.

ConsistencyLevel — Consistency level

"ONE" (default) | character vector | string scalar

Consistency level, specified as one of these values.

Consistency Level Value	Consistency Level Description
"ALL"	Return query results when all replica nodes respond (read operation) or commit the change (write operation).

Consistency Level Value	Consistency Level Description
"EACH_QUORUM"	Finish execution when most replica nodes in each data center commit the change (write operation only).
"QUORUM"	Return query results when most replica nodes respond (read operation) or commit the change (write operation).
"LOCAL_QUORUM"	Return query results when most replica nodes in the local data center respond (read operation) or commit the change (write operation).
"ONE" (default)	Return query results when one replica node responds (read operation) or commits the change (write operation).
"TWO"	Return query results when two replica nodes respond (read operation) or commit the change (write operation).
"THREE"	Return query results when three replica nodes respond (read operation) or commit the change (write operation).
"LOCAL_ONE"	Return query results when one replica node in the local data center responds (read operation) or commits the change (write operation).
"ANY"	Finish execution even if all replica nodes for the specified partition are not available (write operation only).
"SERIAL"	Return query results for current (and possibly uncommitted) data for replica nodes in any data center (read operation only).
"LOCAL_SERIAL"	Return query results for current (and possibly uncommitted) data for replica nodes in the local data center (read operation only).

You can specify the value of the consistency level as a character vector or string scalar.

For details about consistency levels, see [Configuring Data Consistency](#).

Data Types: `char` | `string`

RequestTimeout – Request timeout

12 (default) | positive numeric scalar

This property is read-only.

Request timeout, specified as a positive numeric scalar. The request timeout indicates the number of seconds the database waits to return a CQL query before throwing an error.

Data Types: `double`

Output Arguments

results – CQL query results

table

CQL query results, returned as a table. Each Cassandra database column from the result of the CQL query is a variable in the table. The variable names match the names of the Cassandra database columns from the result of the CQL query.

The data types of the variables in the table depend on the CQL data types. For details about how CQL data types convert to MATLAB data types, see “Convert CQL Data Types to MATLAB Data Types Using Apache Cassandra Database C++ Interface” on page 8-2.

For read or write operations that return no data, the `executecql` function returns an empty table.

Version History

Introduced in R2021a

See Also

Objects

`connection`

Functions

`apacheCassandra` | `tablenames` | `columninfo` | `partitionRead` | `upsert` | `close`

Topics

“Import Data from Cassandra Database Table Using CQL” on page 8-14

“Convert CQL Data Types to MATLAB Data Types Using Apache Cassandra Database C++ Interface” on page 8-2

“Fill Values for Missing Data from Database Using Apache Cassandra Database C++ Interface” on page 8-9

External Websites

[Apache Cassandra Documentation](#)

[CQL Reference Documentation](#)

[CQL Data Types](#)

[Configuring Data Consistency](#)

