

Database Toolbox™

User's Guide



MATLAB®

R2018a

 MathWorks®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Database Toolbox™ User's Guide

© COPYRIGHT 1998–2018 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)
reApril 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)

1 Before You Begin

Database Toolbox Product Description	1-2
Key Features	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

Access Relational Database Data in MATLAB	2-3
Working with MATLAB Interface to SQLite	2-6
MATLAB Interface to SQLite Advantages	2-6
SQLite JDBC Connection Differences	2-6
MATLAB Interface to SQLite Workflow	2-7
MATLAB Interface to SQLite Limitations	2-7
Connection Options	2-9
Creating or Connecting to Data Source	2-9
Defining Operating System Authentication	2-9
Connection Options	2-9
Setup Requirements for Database Connection	2-12

Choosing Between ODBC and JDBC Drivers	2-13
Defining Database Drivers	2-13
Deciding Between ODBC and JDBC Drivers	2-13
Configuring Driver and Data Source	2-15
Microsoft Access ODBC for Windows	2-18
Step 1. Setup the sample Access database.	2-18
Step 2. Verify the driver installation.	2-18
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	2-24
Step 1. Verify the driver installation.	2-24
Step 2. Set up the data source using the Database Explorer app.	2-24
Step 3. Connect using the Database Explorer app or the command line.	2-29
Microsoft SQL Server JDBC for Windows	2-32
Step 1. Verify the driver installation.	2-32
Step 2. Verify the port number.	2-32
Step 3. Set up the operating system authentication.	2-35
Step 4. Add the JDBC driver to the MATLAB static Java class path.	2-36
Step 5. Set up the data source using the Database Explorer app.	2-37
Step 6. Connect using the Database Explorer app or the command line.	2-39
Oracle ODBC for Windows	2-42
Step 1. Verify the driver installation.	2-42
Step 2. Set up the data source using the Database Explorer app.	2-42
Step 3. Connect using the Database Explorer app or the command line.	2-46
Oracle JDBC for Windows	2-48
Step 1. Verify the driver installation.	2-48
Step 2. Set up the operating system authentication.	2-48

Step 3. Add the JDBC driver to the MATLAB static Java class path.	2-49
Step 4. Set up the data source using the Database Explorer app.	2-49
Step 5. Connect using the Database Explorer app or the command line.	2-52
MySQL ODBC for Windows	2-55
Step 1. Verify the driver installation.	2-55
Step 2. Set up the data source using the Database Explorer app.	2-55
Step 3. Connect using the Database Explorer app or the command line.	2-59
MySQL JDBC for Windows	2-61
Step 1. Verify the driver installation.	2-61
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-61
Step 3. Set up the data source using the Database Explorer app.	2-62
Step 4. Connect using the Database Explorer app or the command line.	2-64
PostgreSQL ODBC for Windows	2-66
Step 1. Verify the driver installation.	2-66
Step 2. Set up the data source using the Database Explorer app.	2-66
Step 3. Connect using the Database Explorer app or the command line.	2-70
PostgreSQL JDBC for Windows	2-72
Step 1. Verify the driver installation.	2-72
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-72
Step 3. Set up the data source using the Database Explorer app.	2-73
Step 4. Connect using the Database Explorer app or the command line.	2-75
SQLite JDBC for Windows	2-77
Step 1. Verify the driver installation.	2-77
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-77

Step 3. Set up the data source using the Database Explorer app.	2-78
Step 4. Connect using the Database Explorer app or the command line.	2-80
Microsoft SQL Server JDBC for macOS	2-83
Step 1. Verify the driver installation.	2-83
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-83
Step 3. Set up the data source using the Database Explorer app.	2-84
Step 4. Connect using the Database Explorer app or the command line.	2-86
Microsoft SQL Server JDBC for Linux	2-88
Step 1. Verify the driver installation.	2-88
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-88
Step 3. Set up the data source using the Database Explorer app.	2-89
Step 4. Connect using the Database Explorer app or the command line.	2-91
Oracle JDBC for macOS	2-94
Step 1. Verify the driver installation.	2-94
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-94
Step 3. Set up the data source using the Database Explorer app.	2-95
Step 4. Connect using the Database Explorer app or the command line.	2-97
Oracle JDBC for Linux	2-100
Step 1. Verify the driver installation.	2-100
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-100
Step 3. Set up the data source using the Database Explorer app.	2-101
Step 4. Connect using the Database Explorer app or the command line.	2-103
MySQL JDBC for macOS	2-106
Step 1. Verify the driver installation.	2-106

Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-106
Step 3. Set up the data source using the Database Explorer app.	2-107
Step 4. Connect using the Database Explorer app or the command line.	2-109
MySQL JDBC for Linux	2-111
Step 1. Verify the driver installation.	2-111
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-111
Step 3. Set up the data source using the Database Explorer app.	2-112
Step 4. Connect using the Database Explorer app or the command line.	2-114
PostgreSQL JDBC for macOS	2-116
Step 1. Verify the driver installation.	2-116
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-116
Step 3. Set up the data source using the Database Explorer app.	2-117
Step 4. Connect using the Database Explorer app or the command line.	2-119
PostgreSQL JDBC for Linux	2-122
Step 1. Verify the driver installation.	2-122
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-122
Step 3. Set up the data source using the Database Explorer app.	2-123
Step 4. Connect using the Database Explorer app or the command line.	2-125
SQLite JDBC for macOS	2-127
Step 1. Verify the driver installation.	2-127
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-127
Step 3. Set up the data source using the Database Explorer app.	2-128
Step 4. Connect using the Database Explorer app or the command line.	2-130

SQLite JDBC for Linux	2-133
Step 1. Verify the driver installation.	2-133
Step 2. Add the JDBC driver to the MATLAB static Java class path.	2-133
Step 3. Set up the data source using the Database Explorer app.	2-134
Step 4. Connect using the Database Explorer app or the command line.	2-136
Other ODBC-Compliant or JDBC-Compliant Databases	2-139
ODBC-Compliant Databases	2-139
JDBC-Compliant Databases	2-139
Connecting to Database	2-142
Microsoft Access	2-142
Microsoft SQL Server	2-142
Oracle	2-142
MySQL	2-143
PostgreSQL	2-143
SQLite	2-143
Other ODBC-Compliant or JDBC-Compliant Databases	2-144
Data Import Using Database Explorer App or Command Line	2-145
Data Import Using Database Explorer App	2-145
Data Import Using Command Line	2-146
Custom Data Types	2-147
SQL Queries Saved in Scripts or Files	2-147
Inserting Data Using Command Line	2-148
Working with Large Data Sets	2-150
Connect to a Database with Maximum Performance	2-150
Import Large Data Sets into MATLAB	2-150
Export Large Data Sets from MATLAB	2-151
Access Large Data Using a DatabaseDatastore	2-151
Deploying Database Application with MATLAB Compiler ..	2-152
Create and Deploy Database Application	2-152
About Driver Configurations	2-152
Working with Database Toolbox Preferences	2-154

Writing Data Common Errors	3-2
Fetching Data Common Errors	3-4
Database Connection Error Messages	3-9
Database Explorer App Error Messages	3-16
Connecting to Database Using Native ODBC Interface	3-19
About Native ODBC Interface	3-19
Native ODBC Interface Workflow	3-19
Database Connection Type Comparison	3-21

Create SQL Queries Using Database Explorer App	4-2
Create SQL Query Using Toolstrip Buttons	4-2
Enter SQL Query Manually	4-3
Work with Multiple SQL Queries	4-4
SQL Query Limitations	4-5
Join Tables Using Database Explorer App	4-6
Different Join Types	4-6
Join Tables	4-6
Join Diagram	4-9
Join Type Limitations	4-9
Data Preview Using Database Explorer App	4-11
Automatic Preview	4-11
Preview Size	4-11
Preview Data by Creating SQL Query	4-12
Preview Data by Entering SQL Query Manually	4-12
Modify and Delete Data Sources	4-14
Modify Data Sources	4-14

Delete Data Sources	4-15
Generate SQL Query and MATLAB Script	4-17
Generate SQL Query	4-17
Generate MATLAB Script	4-18

Using Database Toolbox Functions

5

Import Data from Databases into MATLAB	5-3
Create Queries with Characters and Variables	5-7
Create Query Using Date	5-7
Create Query Using Text	5-8
Create Query Using MATLAB Variable	5-9
Create Query Using Special Characters	5-10
Roll Back and Commit Data in Database	5-12
Change Database Connection Catalog	5-13
Create Table and Add Column	5-14
Delete Data from Databases	5-15
Roll Back Data After Updating Record	5-18
Export Data to New Record in Database	5-21
Replace Existing Data in Database	5-25
Export Multiple Records from MATLAB Workspace	5-27
Export Data Using Bulk Insert	5-32
Bulk Insert Functionality	5-32
Bulk Insert into Oracle	5-32
Bulk Insert into Microsoft SQL Server 2005	5-34
Bulk Insert into MySQL	5-35
Display Database Metadata	5-38

Call Stored Procedure That Returns Data	5-41
Run Custom Database Function	5-44
Data Import Approaches and Memory Management	5-46
Data Import in One Step	5-46
Data Import in Two Steps	5-48
Large Data Import Using Row Limits	5-48
Display Information About Imported Data	5-50
Using Scrollable Cursors	5-53
Scrollable Cursors	5-53
Differences Between Native ODBC and JDBC	
Scrollable Cursors	5-54
Import Data Using Scrollable Cursor with Relative Position Off set	5-61
Import Large Data Using DatabaseDatastore Object	5-64
Import Data Using MATLAB® Interface to SQLite	5-68
Retrieve Image Data Types	5-73
Import Boolean Data from Database	5-78
Append Data to Existing Database Table Using Insert Functionality	5-80
Insert Data into New Database Table Using Insert Functionality	5-82
Join Tables Using Command Line	5-84
Import Data from Database Table Using sqlread Function ..	5-85
Insert Data into Database Table	5-88
Retrieve Database Metadata	5-91

6

Explore Graph Database Structure	6-2
Working with MATLAB Interface to Neo4j	6-9
About Neo4j Graph Databases	6-9
MATLAB Interface to Neo4j Workflow	6-9
Searching Graph Database Using MATLAB Interface to Neo4j	6-12
MATLAB Interface to Neo4j Search Functions	6-12
General and Targeted Search Workflows	6-12
Update Friend Information in Social Neighborhood	6-15
Add and Query Group of Colleagues in Social Neighborhood	6-20
MATLAB Interface to Neo4j Error Messages	6-28

Database Toolbox Interface For MongoDB Topics

7

Import and Analyze Data from MongoDB	7-2
Import Filtered Data from MongoDB	7-5
Import Large Data from MongoDB	7-8
Export MATLAB Data into MongoDB	7-11
Import and Export MATLAB Objects Using MongoDB	7-15
Database Toolbox Interface for MongoDB Installation	7-19
Installation	7-19
Supported MongoDB Versions	7-19
Database Toolbox Interface for MongoDB Error Messages ..	7-21

Before You Begin

- “Database Toolbox Product Description” on page 1-2
- “Data Type Support” on page 1-3
- “Data Retrieval Restrictions” on page 1-5

Database Toolbox Product Description

Exchange data with relational and nonrelational databases

Database Toolbox provides functions and an app for working with relational databases. It includes support for nonrelational databases, and provides a native SQLite database. You can access data in relational databases using SQL commands, or use the Database Explorer app to interact with a database without using SQL.

The toolbox can connect to any ODBC- or JDBC-compliant relational database, including Oracle®, SAS®, MySQL®, Microsoft® SQL Server®, Microsoft Access™, and PostgreSQL. You can create, query, and manipulate native SQLite relational databases without additional software or database drivers.

The toolbox supports nonrelational databases Neo4j® and MongoDB®. The Neo4j interface lets you access data stored as graphs or queried using nongraph operations. The NoSQL database interface to MongoDB provides access to unstructured data.

The toolbox lets you access multiple databases simultaneously within a single session and enables segmented import of large data sets using DatabaseDatastore. You can split SQL queries and parallelize access to data (with MATLAB® Distributed Computing Server™ and Parallel Computing Toolbox™).

Key Features

- Database Explorer app for working with databases interactively
- JDBC- and ODBC-compliant database connections, with fast read/write via a native ODBC interface
- Functions for executing queries using SQL files and SQL statements
- Data import and export with multiple databases in a single session
- Large data set import via a single transaction, via multiple transactions, or as a DatabaseDatastore object
- Direct data import into numeric, cell, structure, table, and dataset arrays
- Support for NoSQL databases MongoDB and Neo4j

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 <code>TIMESTAMP</code> data into MATLAB, you can get an incorrect value at the daylight savings time change. To avoid an incorrect value, convert <code>TIMESTAMP</code> 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>

Database	Supported Data Type
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

`datainsert` | `exec` | `fastinsert` | `fetch` | `insert` | `update`

More About

- “Connecting to Database Using Native ODBC Interface” on page 3-19
- “Import Data from Databases into MATLAB” on page 5-3
- “Export Data to New Record in Database” on page 5-21

Data Retrieval Restrictions

In this section...
“Spaces in Table Names or Column Names” on page 1-5
“Quotation Marks in Table Names or Column Names” on page 1-5
“Reserved Words in Column Names” on page 1-5

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 2-145

Getting Started with Database Toolbox

- “Access Relational Database Data in MATLAB” on page 2-3
- “Working with MATLAB Interface to SQLite” on page 2-6
- “Connection Options” on page 2-9
- “Setup Requirements for Database Connection” on page 2-12
- “Choosing Between ODBC and JDBC Drivers” on page 2-13
- “Configuring Driver and Data Source” on page 2-15
- “Microsoft Access ODBC for Windows” on page 2-18
- “Microsoft SQL Server ODBC for Windows” on page 2-24
- “Microsoft SQL Server JDBC for Windows” on page 2-32
- “Oracle ODBC for Windows” on page 2-42
- “Oracle JDBC for Windows” on page 2-48
- “MySQL ODBC for Windows” on page 2-55
- “MySQL JDBC for Windows” on page 2-61
- “PostgreSQL ODBC for Windows” on page 2-66
- “PostgreSQL JDBC for Windows” on page 2-72
- “SQLite JDBC for Windows” on page 2-77
- “Microsoft SQL Server JDBC for macOS” on page 2-83
- “Microsoft SQL Server JDBC for Linux” on page 2-88
- “Oracle JDBC for macOS” on page 2-94
- “Oracle JDBC for Linux” on page 2-100
- “MySQL JDBC for macOS” on page 2-106
- “MySQL JDBC for Linux” on page 2-111
- “PostgreSQL JDBC for macOS” on page 2-116
- “PostgreSQL JDBC for Linux” on page 2-122

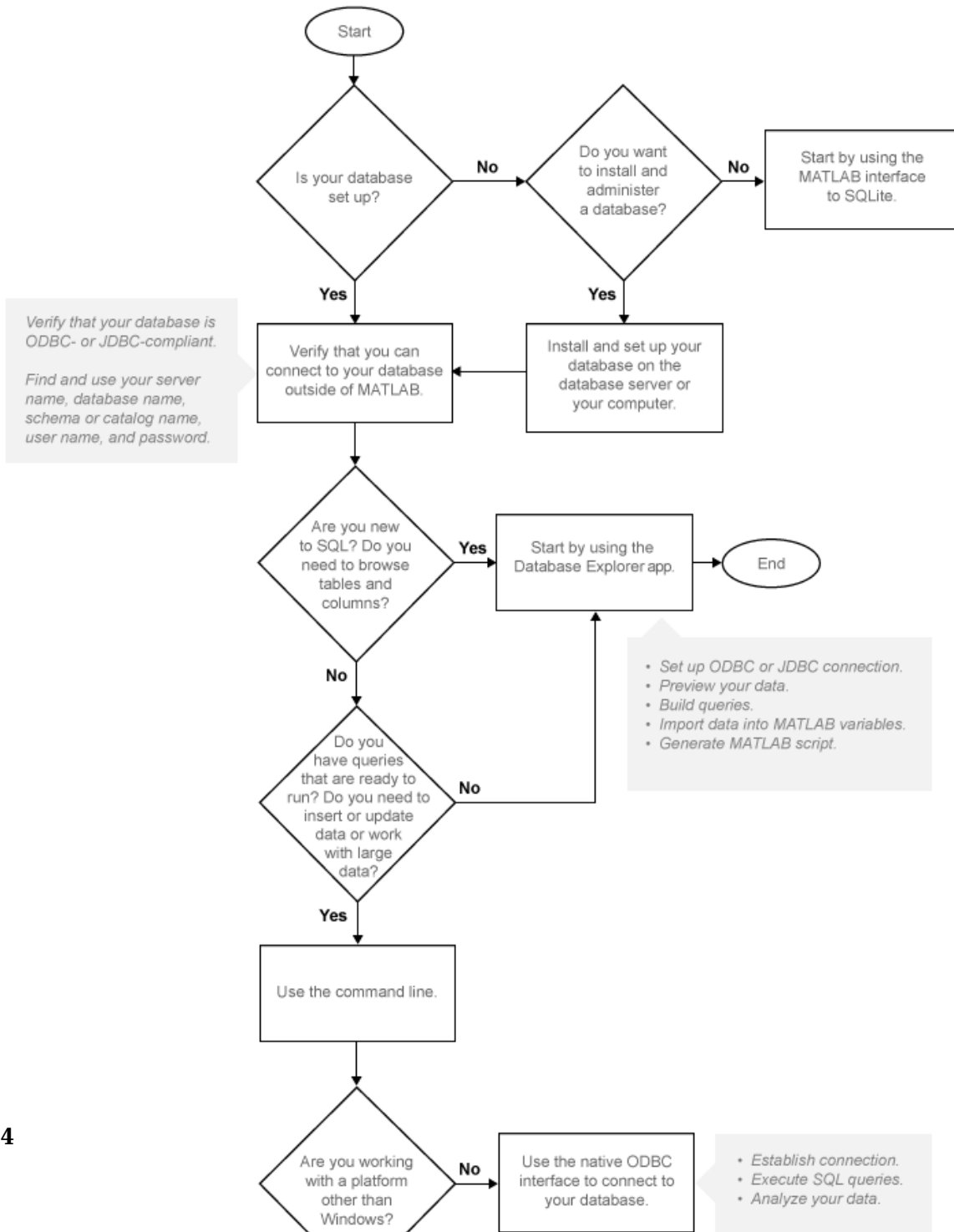
- “SQLite JDBC for macOS” on page 2-127
- “SQLite JDBC for Linux” on page 2-133
- “Other ODBC-Compliant or JDBC-Compliant Databases” on page 2-139
- “Connecting to Database” on page 2-142
- “Data Import Using Database Explorer App or Command Line” on page 2-145
- “Inserting Data Using Command Line” on page 2-148
- “Working with Large Data Sets” on page 2-150
- “Deploying Database Application with MATLAB Compiler” on page 2-152
- “Working with Database Toolbox Preferences” on page 2-154

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 “Working with MATLAB Interface to SQLite” on page 2-6.
- 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.
- 4** Choose between using an ODBC driver or a JDBC driver. For details, see “Choosing Between ODBC and JDBC Drivers” on page 2-13.
- 5** For ODBC drivers, the driver is typically preinstalled on your computer. For JDBC drivers, install the driver. 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.
- 6** Create a data source for ODBC-compliant drivers. For JDBC-compliant drivers, add the full path of the driver to the static Java® class path. Then, create a data source for JDBC-compliant drivers. For details, see “Configuring Driver and Data Source” on page 2-15.
- 7** Test the connection to your database using the Database Explorer app or the command line.
- 8** Connect to your database using the Database Explorer app or the command line. For details, see “Connecting to Database” on page 2-142.
- 9** 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 2-145.
- 10** Insert data into your database by exporting data from a MATLAB variable. For details, see “Inserting Data Using Command Line” on page 2-148.
- 11** 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 4-18.

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 2-12
- “Choosing Between ODBC and JDBC Drivers” on page 2-13
- “Configuring Driver and Data Source” on page 2-15
- “Connecting to Database” on page 2-142
- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Working with MATLAB Interface to SQLite” on page 2-6

Working with MATLAB Interface to SQLite

In this section...

“MATLAB Interface to SQLite Advantages” on page 2-6

“SQLite JDBC Connection Differences” on page 2-6

“MATLAB Interface to SQLite Workflow” on page 2-7

“MATLAB Interface to SQLite Limitations” on page 2-7

To analyze your data using SQL in MATLAB without access to a 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 a SQLite database file. The SQLite connection is different from a database connection created using a JDBC driver. For background information about SQLite databases, see SQLite Home Page. To use all the Database Toolbox functionality, install the SQLite JDBC driver and connect to your SQLite database file using a URL string. For details, see “Configuring Driver and Data Source” on page 2-15.

MATLAB Interface to SQLite Advantages

The advantages of using the MATLAB interface to SQLite are:

- Start working with data immediately after installing the Database Toolbox by creating a SQLite database file.
- No installation or administration of software or drivers required.
- Share data using SQLite database files.
- Support for Windows®, Linux®, and Mac.

SQLite JDBC Connection Differences

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

	SQLite Connection Using the MATLAB Interface to SQLite	SQLite Database Connection Using a JDBC Driver
Driver installation	Not required	Required

	SQLite Connection Using the MATLAB Interface to SQLite	SQLite Database Connection Using a JDBC Driver
Database installation	Not required	Required
Database administration	Not required	Required
Database connection function	<code>sqlite</code>	<code>database</code>
Import data	Yes	Yes
Export data	Yes	Yes
Database Explorer	No	Yes
Run stored procedures	No	Yes
Database metadata	No	Yes
Other complex database operations and functionality	No	Yes

MATLAB Interface to SQLite Workflow

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 a SQLite database file using `sqlite`. The SQLite database file has a `.db` extension.
- 2 Create tables in the SQLite database file using `exec`.
- 3 Export your data into the SQLite database file using `insert`.
- 4 Import data into MATLAB using `fetch`.
- 5 Perform data analysis in MATLAB.
- 6 Export results into the SQLite database file using `insert`.
- 7 Close the SQLite connection using `close`.
- 8 Share the SQLite database file with others.

MATLAB Interface to SQLite Limitations

The limitations of using the MATLAB interface to SQLite are:

- Only DOUBLE, INT64, and CHAR data types are supported.
- NULL values in columns are not supported.
- Database Explorer is not supported. Use the command line.

See Also

`close` | `exec` | `fetch` | `insert` | `sqlite`

Related Examples

- “Import Data Using MATLAB® Interface to SQLite” on page 5-68

More About

- “Access Relational Database Data in MATLAB” on page 2-3
- “Configuring Driver and Data Source” on page 2-15

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 “Working with MATLAB Interface to SQLite” on page 2-6.

Creating or Connecting to Data Source

If you already have a database driver installed, you can create a data source. For an ODBC driver, use the Microsoft ODBC Data Source Administrator. For a JDBC driver, add the path of the driver to the Java class path in MATLAB. For examples, see “Configuring Driver and Data Source” on page 2-15. Otherwise, see Driver Installation for information about installing your driver. If your data sources are defined, then you are ready to connect to your database. For details, see “Connecting to Database” on page 2-142. 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 2-145.

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 2-35

Connection Options

Use this table to choose your best connection option.

Connection Option	Why Use This Option?
Database Explorer app	<p>Use the Database Explorer app to:</p> <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	<p>Use the command line to:</p> <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.

There are multiple options to connect to your database using the command line. Use this table to choose your best connection option.

Command-Line Connection Option	Why Use This Option?
ODBC connection	Connect to your database with maximum performance. For details, see “Connecting to Database Using Native ODBC Interface” on page 3-19.
JDBC connection	Achieve maximum platform independence. Use functionality not supported by native ODBC.
SQLite connection	Import data without installing a database or a driver. For details, see “Working with MATLAB Interface to SQLite” on page 2-6.

You can create multiple connections at a time to the same database or a different database. 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

database

More About

- “Choosing Between ODBC and JDBC Drivers” on page 2-13
- “Connecting to Database” on page 2-142
- “Connecting to Database Using Native ODBC Interface” on page 3-19
- “Data Import Using Database Explorer App or Command Line” on page 2-145
- “Working with MATLAB Interface to SQLite” on page 2-6

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 “Working with MATLAB Interface to SQLite” on page 2-6.
- 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 2-3
- “Connecting to Database” on page 2-142

Choosing Between ODBC and JDBC Drivers

In this section...

“Defining Database Drivers” on page 2-13

“Deciding Between ODBC and JDBC Drivers” on page 2-13

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

Use JDBC for:

- Platform independence, allowing you to work with any operating system (including Mac and Linux), driver version, or bitness
- Access to Database Toolbox functions not supported by the native ODBC interface (such as `runstoredprocedure`)

See Also

`close` | `database`

More About

- “Access Relational Database Data in MATLAB” on page 2-3
- “Connection Options” on page 2-9
- “Configuring Driver and Data Source” on page 2-15
- “Connecting to Database” on page 2-142
- “Connecting to Database Using Native ODBC Interface” on page 3-19
- “Working with Large Data Sets” on page 2-150

Configuring Driver and Data Source

Database Toolbox enables you to create a connection to a database installed on a machine or located in the cloud. To connect to an installed database, first install the driver. For all databases, define a data source for ODBC or add the full path of the driver to the static Java class path for JDBC. 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 “Working with MATLAB Interface to SQLite” on page 2-6.

ODBC uses a Data Source Name (DSN) that is the logical name to refer to the drive 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.

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 2-18		
Microsoft SQL Server	“Microsoft SQL Server ODBC for Windows” on page 2-24 “Microsoft SQL Server JDBC for Windows” on page 2-32	“Microsoft SQL Server JDBC for macOS” on page 2-83	“Microsoft SQL Server JDBC for Linux” on page 2-88
Oracle	“Oracle ODBC for Windows” on page 2-42 “Oracle JDBC for Windows” on page 2-48	“Oracle JDBC for macOS” on page 2-94	“Oracle JDBC for Linux” on page 2-100

Database	Platform		
	Windows	macOS 64-bit	Linux 64-bit
MySQL	“MySQL ODBC for Windows” on page 2-55 “MySQL JDBC for Windows” on page 2-61	“MySQL JDBC for macOS” on page 2-106	“MySQL JDBC for Linux” on page 2-111
PostgreSQL	“PostgreSQL ODBC for Windows” on page 2-66 “PostgreSQL JDBC for Windows” on page 2-72	“PostgreSQL JDBC for macOS” on page 2-116	“PostgreSQL JDBC for Linux” on page 2-122
SQLite	“SQLite JDBC for Windows” on page 2-77	“SQLite JDBC for macOS” on page 2-127	“SQLite JDBC for Linux” on page 2-133

Microsoft Access is not supported for Mac 64-bit and Linux 64-bit platforms.

For ODBC-compliant or JDBC-compliant databases that are not listed in the table, see “Other ODBC-Compliant or JDBC-Compliant Databases” on page 2-139.

See Also

close | database

More About

- “Access Relational Database Data in MATLAB” on page 2-3
- “Setup Requirements for Database Connection” on page 2-12
- “Choosing Between ODBC and JDBC Drivers” on page 2-13
- “Connecting to Database” on page 2-142
- “Modify and Delete Data Sources” on page 4-14

- “Working with MATLAB Interface to SQLite” on page 2-6

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 2010 database.

Step 1. Setup the sample Access database.

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

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

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 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](#).

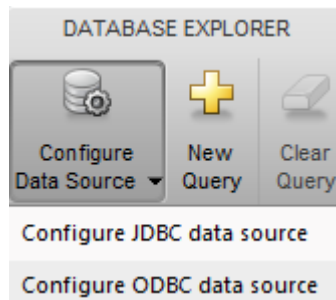
Note Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of Access. Or, to connect to the 32-bit version of Access, see <https://www.mathworks.com/matlabcentral/answers/235949-how-to-connect-to-32-bit-microsoft-access-database-from-64-bit-matlab>.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

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 6.1 for the US English version of Microsoft Access 2010 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.

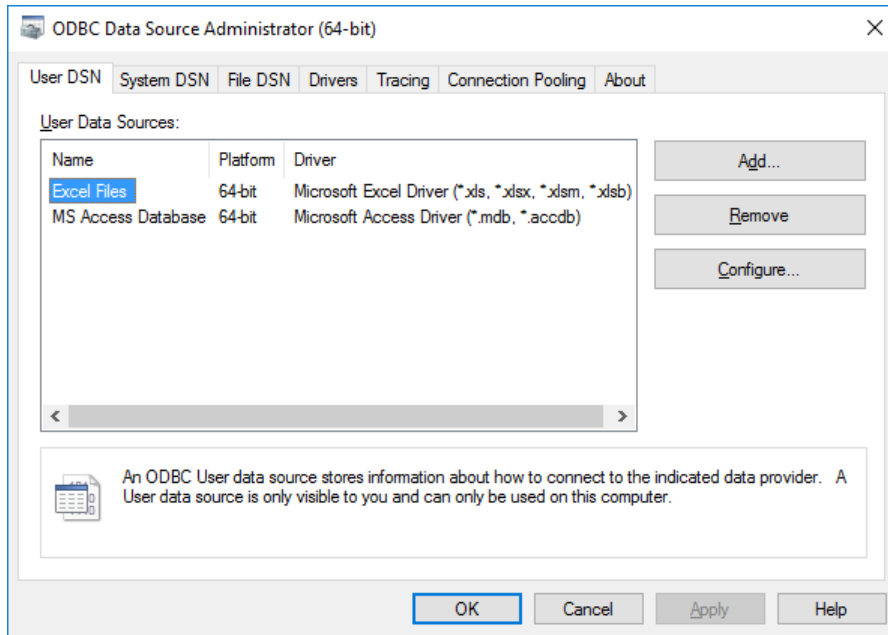
- 1 Close any open Access databases.
- 2 Open the Database Explorer app by clicking the **Apps** tab on the MATLAB Toolstrip. Then, on the right of the **Apps** section, click the **Show more** arrow to open the apps gallery. Under **Database Connectivity and Reporting**, click **Database Explorer**.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



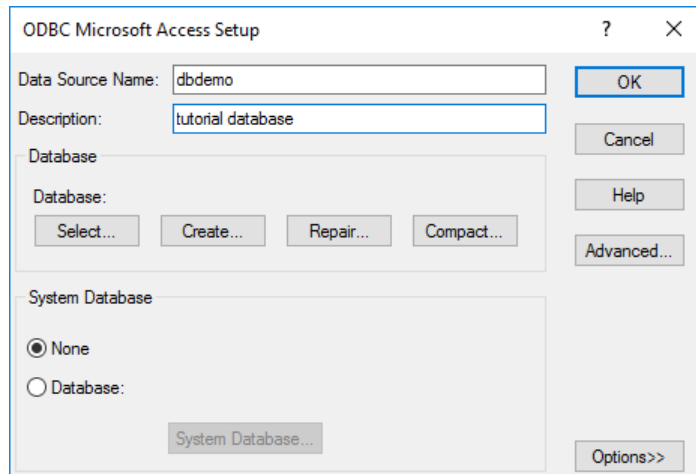
In the ODBC Data Source Administrator dialog box, you define the ODBC data source.

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.mdb`. 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.

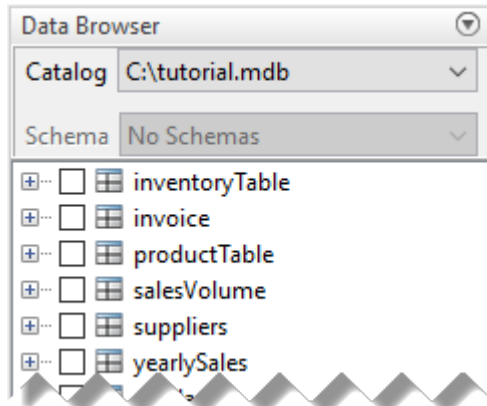
Step 4. Connect using the Database Explorer app or the command line.

Connect to Access Using Database Explorer App

- 1 On the **Database Explorer** tab, in the **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source dbdemo from the **Data Source** list.

- 3 Leave the **Username** and **Password** boxes blank, and click **Connect**.

The Database Explorer 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.



- 4 Select tables in the **Data Browser** pane to query the database.
- 5 Close the **dbdemo** data source tab to close the SQL query and the database connection.

Tip If multiple database connections to different databases are open, then close the Database Explorer app to close all database connections.

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

close | database

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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. This tutorial uses the Microsoft ODBC Driver 13.1 for SQL Server to connect to a Microsoft SQL Server 2016 Express database.

In this section...
“Step 1. Verify the driver installation.” on page 2-24
“Step 2. Set up the data source using the Database Explorer app.” on page 2-24
“Step 3. Connect using the Database Explorer app or the command line.” on page 2-29

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.

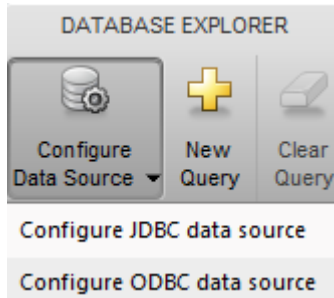
Note Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of SQL Server. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see “Microsoft SQL Server JDBC for Windows” on page 2-32. For details about working with the 64-bit version of Windows, see <https://www.mathworks.com/products/matlab/preparing-for-64-bit-windows.html>.

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.

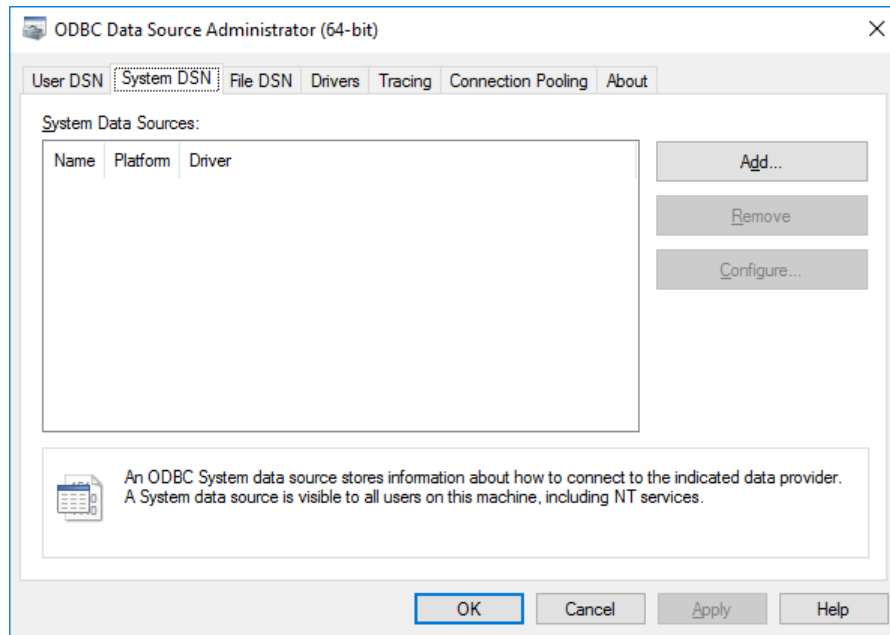
- 2 Select **Configure Data Source > Configure ODBC data source**.



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**.

Microsoft SQL Server

Change the default database to:

toy_store

Mirror server:

SPN for mirror server (Optional):

Attach database filename:

Use ANSI quoted identifiers.

Use ANSI nulls, paddings and warnings.

Application intent:

READWRITE

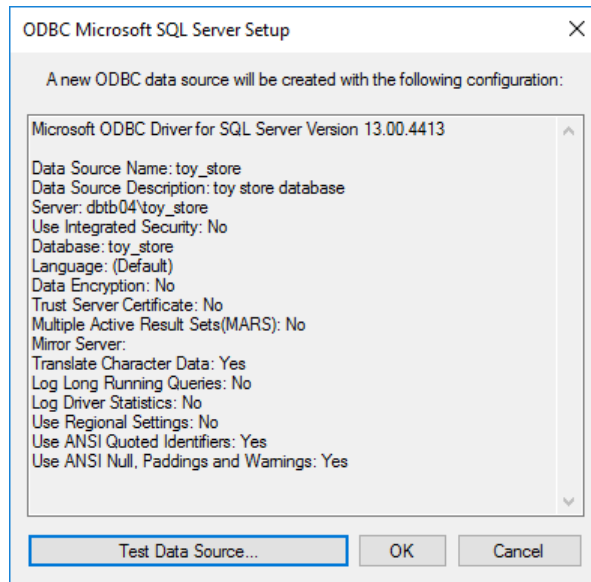
Multi-subnet failover.

Transparent Network IP Resolution.

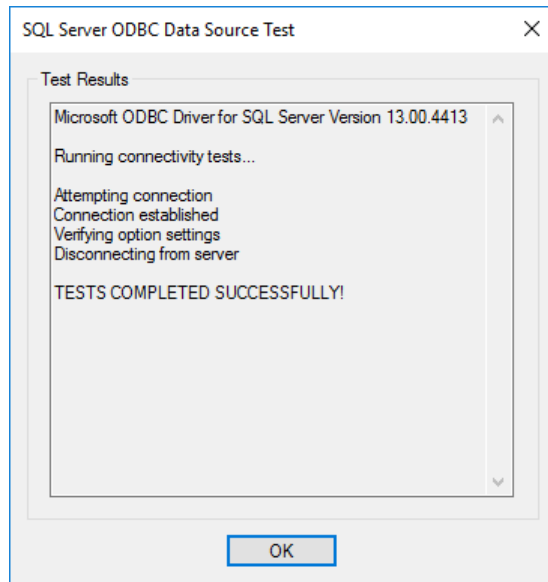
Column Encryption.

< Back Next > Cancel Help

- 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, connect with Windows authentication by selecting the data source you defined from the **Data Source** list. Leave the **Username** and **Password** boxes blank. Click **Connect**.

Or, connect without Windows authentication by selecting the data source you defined. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is MS SQL Server, and two tabs named MS SQL Server and MS SQL Server1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

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

close | database

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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.

In this section...
“Step 1. Verify the driver installation.” on page 2-32
“Step 2. Verify the port number.” on page 2-32
“Step 3. Set up the operating system authentication.” on page 2-35
“Step 4. Add the JDBC driver to the MATLAB static Java class path.” on page 2-36
“Step 5. Set up the data source using the Database Explorer app.” on page 2-37
“Step 6. Connect using the Database Explorer app or the command line.” on page 2-39

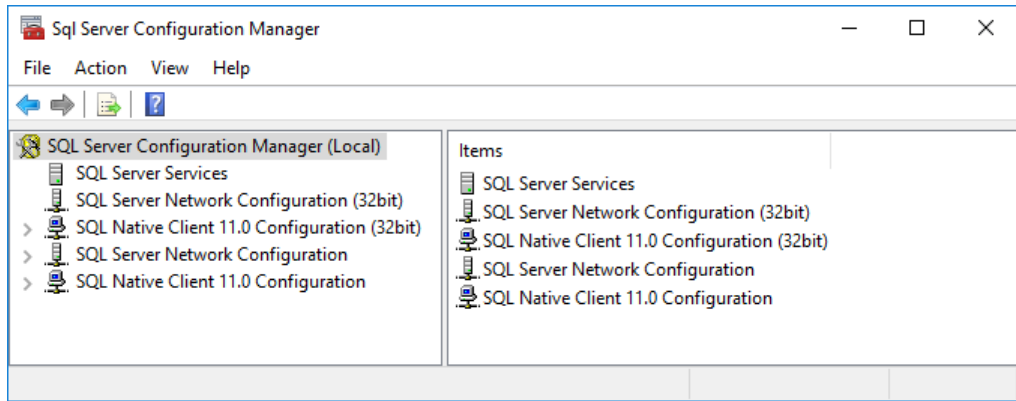
Step 1. Verify the driver installation.

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.

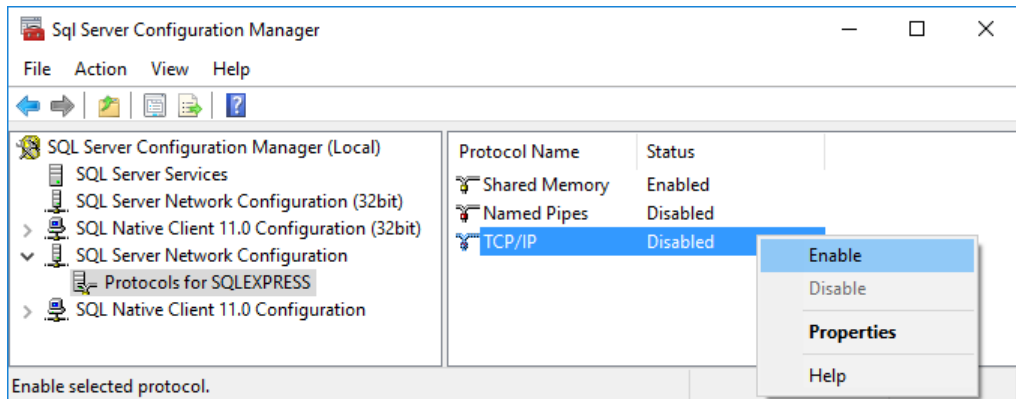
Step 2. Verify the port number.

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.

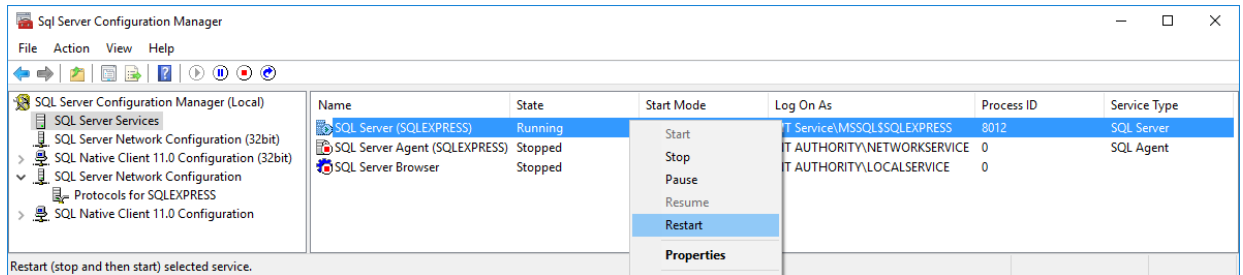
- 1 On the machine where your SQL Server database is installed, click **Start**. Select your Microsoft SQL Server version folder and click **SQL Server Configuration Manager**.



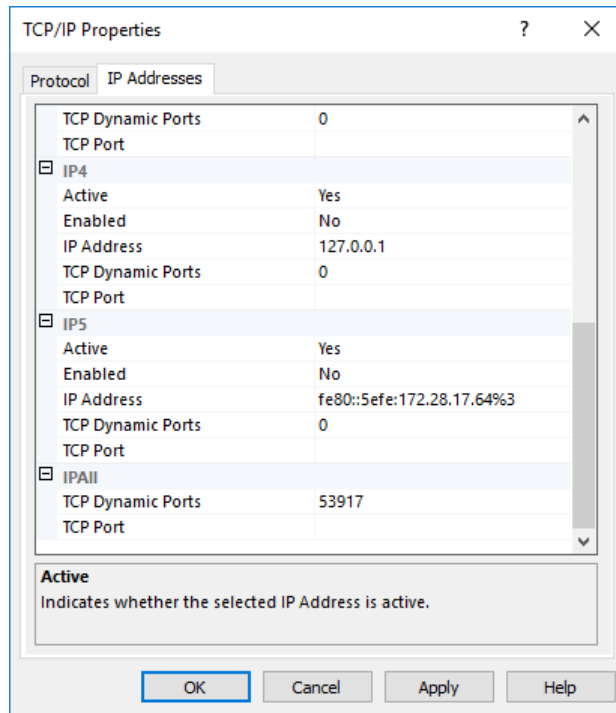
- 2 On the left of the Sql Server Configuration Manager window, click **SQL Server Network Configuration**. Double-click **Protocols for SQLEXPRESS**.
- 3 See if TCP/IP is enabled. If so, skip steps 4 and 5.
- 4 If TCP/IP is disabled, right-click **TCP/IP** and select **Enable**.



- 5 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.



- 6 Click **Protocols for SQLEXPRESS** and right-click **TCP/IP**. Select **Properties**.
- 7 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.



- 8 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” (MATLAB).

- 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 `javaclasspath.txt` in the folder.
- 5 Open `javaclasspath.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 `javaclasspath.txt`.

- 6 Restart MATLAB.

Step 4. Add the JDBC driver to the MATLAB static Java class path.

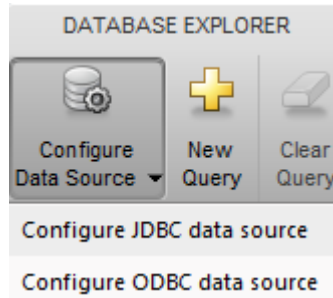
- 1 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaaddpath.txt` in the folder.
- 4 Open `javaaddpath.txt`. Add the full path to the database driver JAR file in `javaaddpath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `C:\DB_Drivers\sqljdbc_4.0\enu\sqljdbc4.jar`. Save and close `javaaddpath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

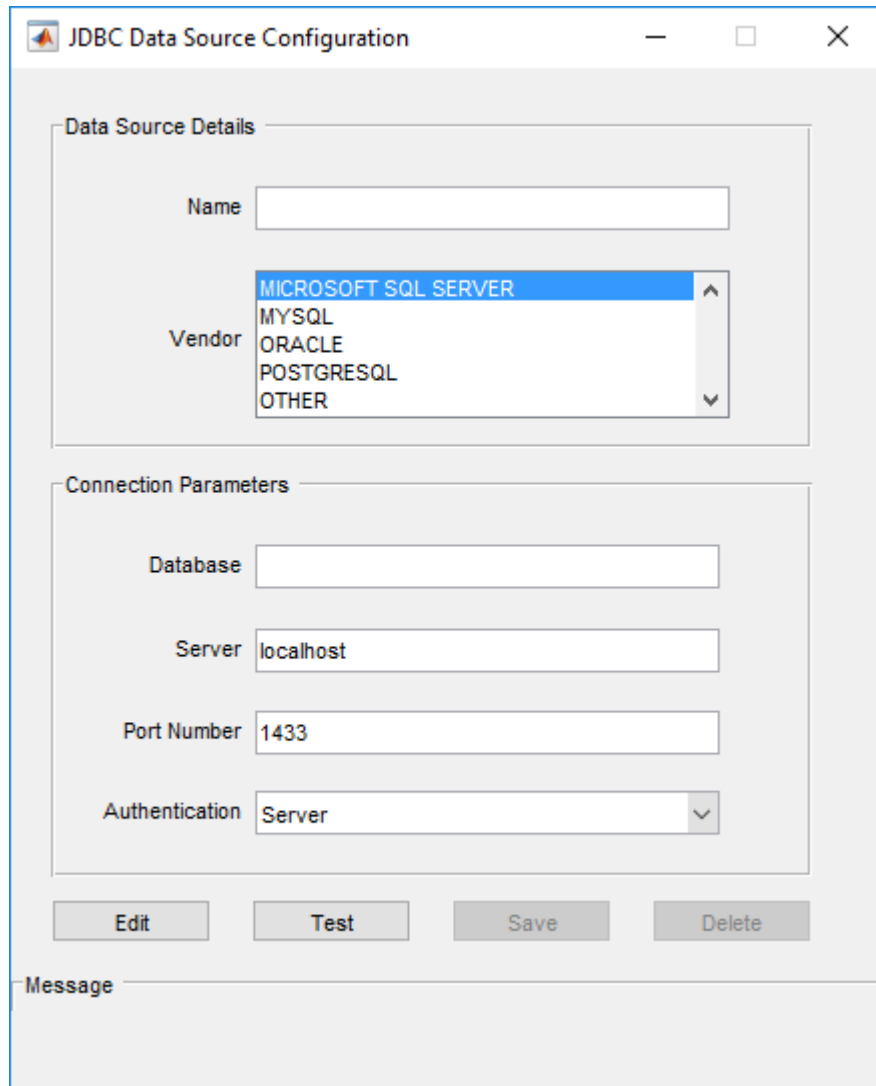
Step 5. Set up the data source using the Database Explorer app.

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to SQL Server Using JDBC Driver and Command Line” on page 2-40.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.



- 3 In the **Name** box, enter a data source name. If you enter a name that duplicates an existing ODBC data source name, the Database Explorer app appends `_JDBC` to the name after you save it.
- 4 From the **Vendor** list, select `MICROSOFT SQL SERVER`. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the

bottom: Unable to find JDBC driver file on MATLAB Java class path. Address this message by following the steps described in Step 4 on page 2-36.

- 5 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.
- 6 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.

- 7 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.

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

Step 6. 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, connect with Windows authentication by selecting the data source you defined from the **Data Source** list. Leave the **Username** and **Password** boxes blank. Click **Connect**.

Or, connect to your database without Windows authentication by selecting the data source you defined. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is MS SQL Server, and two tabs named MS SQL Server and MS SQL Server1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to SQL Server Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 To connect with Windows authentication, use the **Vendor** name-value pair argument of the **database** function to specify a connection to a SQL Server database. Use the **AuthType** name-value pair argument to connect with Windows authentication. Specify a blank user name and password. For example, this code assumes that you are connecting to a database named **dbname**, on a database server named **sname**, with the port number 123456.

```
conn = database('dbname', '', '', 'Vendor', 'Microsoft SQL Server', ...  
              'Server', 'sname', 'AuthType', 'Windows', ...  
              'PortNumber', 123456);
```

Or, to connect without Windows authentication, use the **AuthType** name-value pair argument of the **database** function to specify a connection to the database server. For example, this code assumes that you are connecting to a database named **dbname** with the user name **username** and the password **pwd**.

```
conn = database('dbname', 'username', 'pwd', ...  
              'Vendor', 'Microsoft SQL Server', 'Server', 'sname', ...  
              'AuthType', 'Server', 'PortNumber', 123456);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

close | database | javaaddpath

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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.

In this section...
“Step 1. Verify the driver installation.” on page 2-42
“Step 2. Set up the data source using the Database Explorer app.” on page 2-42
“Step 3. Connect using the Database Explorer app or the command line.” on page 2-46

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.

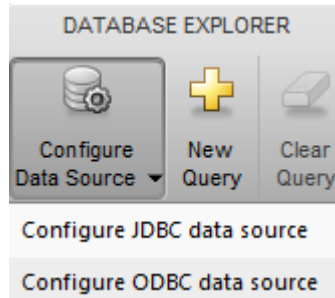
Note: Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of Oracle. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see “Oracle JDBC for Windows” on page 2-48. For details about working with the 64-bit version of Windows, see <https://www.mathworks.com/products/matlab/preparing-for-64-bit-windows.html>.

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.

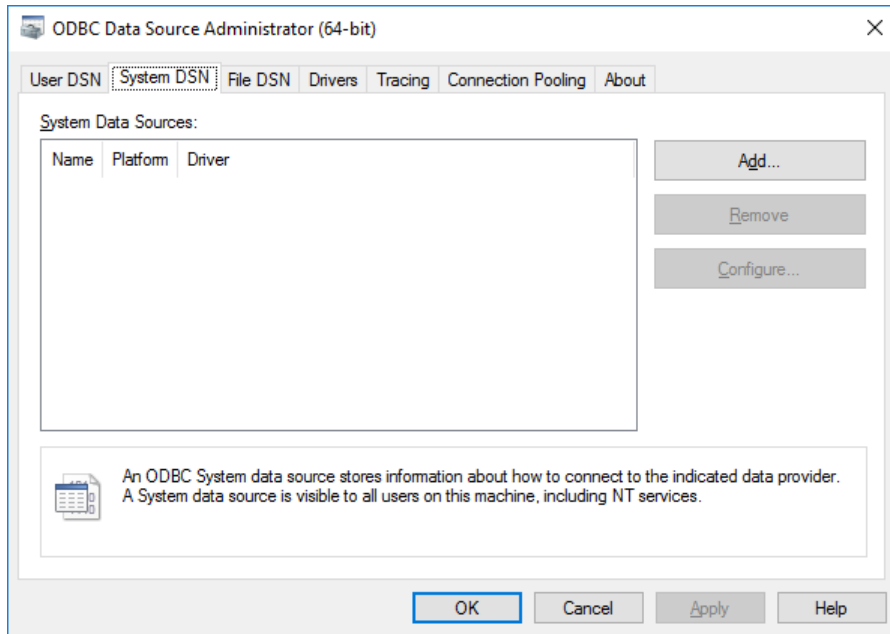
- 2 Select **Configure Data Source > Configure ODBC data source**.



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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, connect with Windows authentication by selecting the data source you defined from the **Data Source** list. Leave the **Username** and **Password** boxes blank. Click **Connect**.

Or, connect without Windows authentication by selecting the data source you defined. Enter a user name and password. Click **Connect**.

The Catalog and Schema dialog box opens.

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

The Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is **ORA**, and two tabs named **ORA** and **ORA1** are open, then close both tabs.

To close all database connections, close the Database Explorer app.

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

`close` | `database`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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.

In this section...
“Step 1. Verify the driver installation.” on page 2-48
“Step 2. Set up the operating system authentication.” on page 2-48
“Step 3. Add the JDBC driver to the MATLAB static Java class path.” on page 2-49
“Step 4. Set up the data source using the Database Explorer app.” on page 2-49
“Step 5. Connect using the Database Explorer app or the command line.” on page 2-52

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” (MATLAB).

- 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 `javalibrarypath.txt` in the folder.
- 5 Open `javalibrarypath.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 `javalibrarypath.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. Add the JDBC driver to the MATLAB static Java class path.

- 1 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`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `C:\DB_Drivers\ojdbc6.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

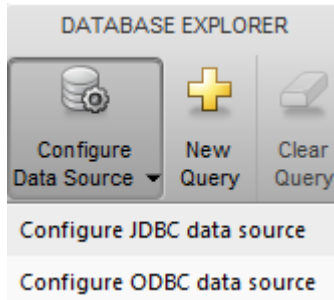
Step 4. Set up the data source using the Database Explorer app.

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to Oracle Using JDBC Driver and Command Line” on page 2-53.

- 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 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. If you enter a name that duplicates an existing ODBC data source name, the Database Explorer app appends `_JDBC` to the name after you save it.
- 4 From the **Vendor** list, select `ORACLE`. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: `Unable to find JDBC driver file on MATLAB Java class path`. Address this message by following the steps described in Step 3 on page 2-49.

JDBC Data Source Configuration

Data Source Details

Name: ORA

Vendor: ORACLE

Connection Parameters

Database:

Server: localhost

Port Number: 1521

Driver Type: thin

Edit Test Save Delete

Message

- 5 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.
- 6 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**.

- 7 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.

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

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

Step 5. 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, connect to your database with Windows authentication by selecting the data source you defined from the **Data Source** list. Leave the **Username** and **Password** boxes blank. Click **Connect**.

Or, connect to your database without Windows authentication by selecting the data source you defined. Enter a user name and password. Click **Connect**.

The Catalog and Schema dialog box opens.

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

The Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is ORA, and two tabs named ORA and ORA1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to Oracle Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 To connect with Windows authentication, use the `Vendor` name-value pair argument of the `database` function to specify a connection to an Oracle database. Use the `DriverType` name-value pair argument to connect with Windows authentication by specifying the `oci` value. Specify a blank user name and password. For example, this code assumes that you are connecting to a database named `dbname`, on a database server named `sname`, with the port number 123456.

```
conn = database('dbname', '', '', ...
               'Vendor', 'Oracle', 'DriverType', 'oci', ...
               'Server', 'sname', 'PortNumber', 123456);
```

`dbname` 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 in `<ORACLE_HOME>\NETWORK\ADMIN` where `<ORACLE_HOME>` is the folder containing the installed database or Oracle client.

Or, to connect without Windows authentication, use the `DriverType` name-value pair argument of the `database` function with the value `thin` to specify a connection to the database server. For example, this code assumes that you are connecting to a database named `dbname` with the user name `username` and the password `pwd`.

```
conn = database('dbname', 'username', 'pwd', ...
               'Vendor', 'Oracle', 'DriverType', 'thin', ...
               'Server', 'sname', 'PortNumber', 123456);
```

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

close | database | javaaddpath

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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. This tutorial uses a MySQL ODBC 5.3 Driver to connect to the MySQL database.

In this section...

“Step 1. Verify the driver installation.” on page 2-55

“Step 2. Set up the data source using the Database Explorer app.” on page 2-55

“Step 3. Connect using the Database Explorer app or the command line.” on page 2-59

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.

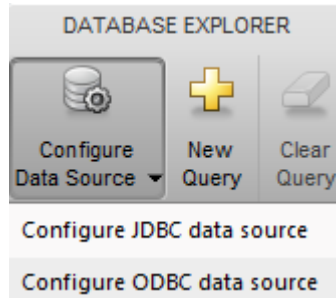
Note: Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of MySQL. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see “MySQL JDBC for Windows” on page 2-61. For details about working with the 64-bit version of Windows, see <https://www.mathworks.com/products/matlab/preparing-for-64-bit-windows.html>.

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.

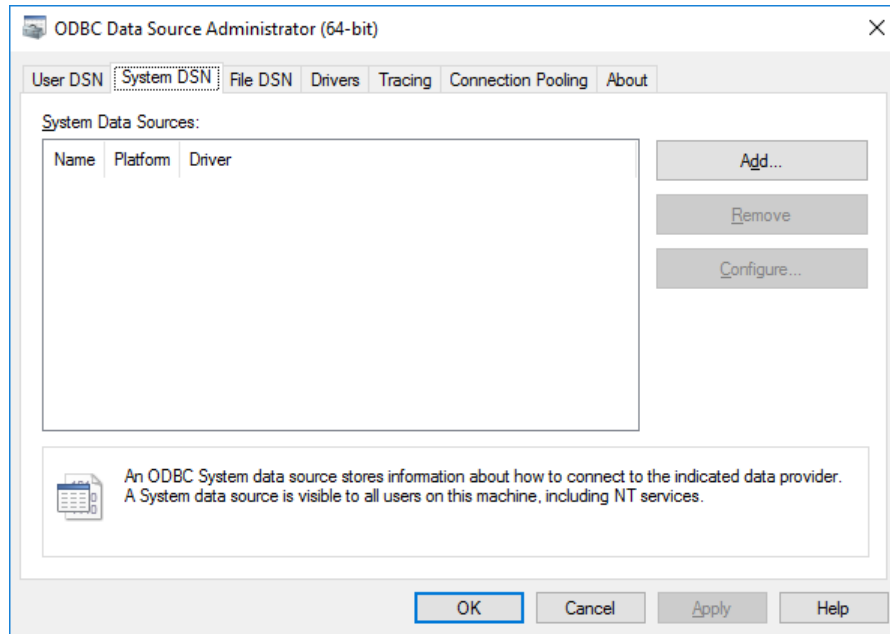
- 2 Select **Configure Data Source > Configure ODBC data source**.



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**.



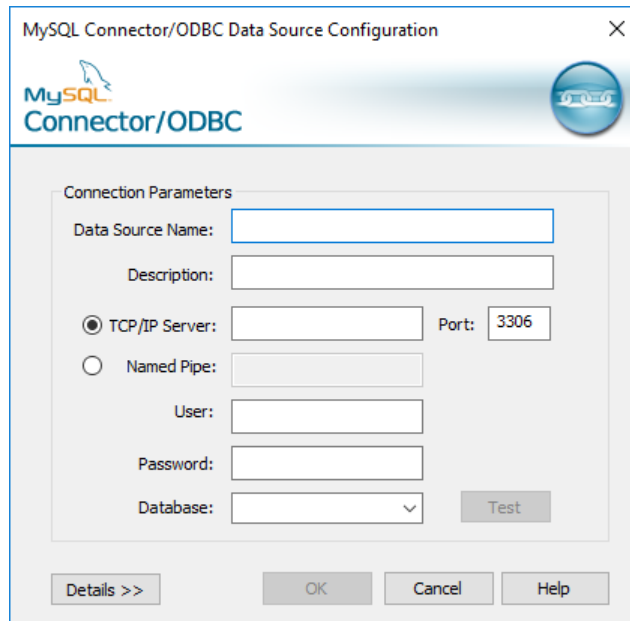
The Create New Data Source dialog box opens and displays a list of installed ODBC drivers.

- 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is MySQL, and two tabs named MySQL and MySQL1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

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

close | database

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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 5.1.17 driver to connect to a MySQL Version 5.5.16 database.

In this section...

“Step 1. Verify the driver installation.” on page 2-61

“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-61

“Step 3. Set up the data source using the Database Explorer app.” on page 2-62

“Step 4. Connect using the Database Explorer app or the command line.” on page 2-64

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. Add the JDBC driver to the MATLAB static Java class path.

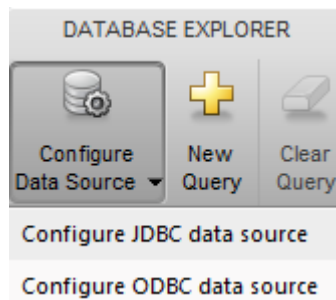
- 1 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `C:\DB_Drivers\mysql-connector-java-5.1.17-bin.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to MySQL Using JDBC Driver and Command Line” on page 2-65.

- 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 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. If you enter a name that duplicates an existing ODBC data source name, the Database Explorer app appends `_JDBC` to the name after you save it.
- 4 From the **Vendor** list, select `MYSQL`. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: `Unable to find JDBC driver file on MATLAB Java class path`. Address this message by following the steps described in Step 2 on page 2-61.

JDBC Data Source Configuration

Data Source Details

Name: MySQL

Vendor: MICROSOFT SQL SERVER, **MYSQL**, ORACLE, POSTGRESQL, OTHER

Connection Parameters

Database:

Server: localhost

Port Number: 3306

Edit Test Save Delete

Message

- 5 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.

- 6 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.

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

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

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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is MySQL, and two tabs named MySQL and MySQL1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to MySQL Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Use the Vendor name-value pair argument of the `database` function to specify a connection to a MySQL database. 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` and the password `pwd`.

```
conn = database('dbname','username','pwd', ...  
              'Vendor','MySQL', ...  
              'Server','sname');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`close` | `database` | `javaaddpath`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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.

In this section...
“Step 1. Verify the driver installation.” on page 2-66
“Step 2. Set up the data source using the Database Explorer app.” on page 2-66
“Step 3. Connect using the Database Explorer app or the command line.” on page 2-70

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.

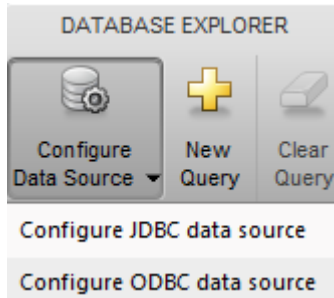
Note: Database Toolbox no longer supports connecting to a database using a 32-bit driver. Use the 64-bit version of PostgreSQL. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see “PostgreSQL JDBC for Windows” on page 2-72. For details about working with the 64-bit version of Windows, see <https://www.mathworks.com/products/matlab/preparing-for-64-bit-windows.html>.

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.

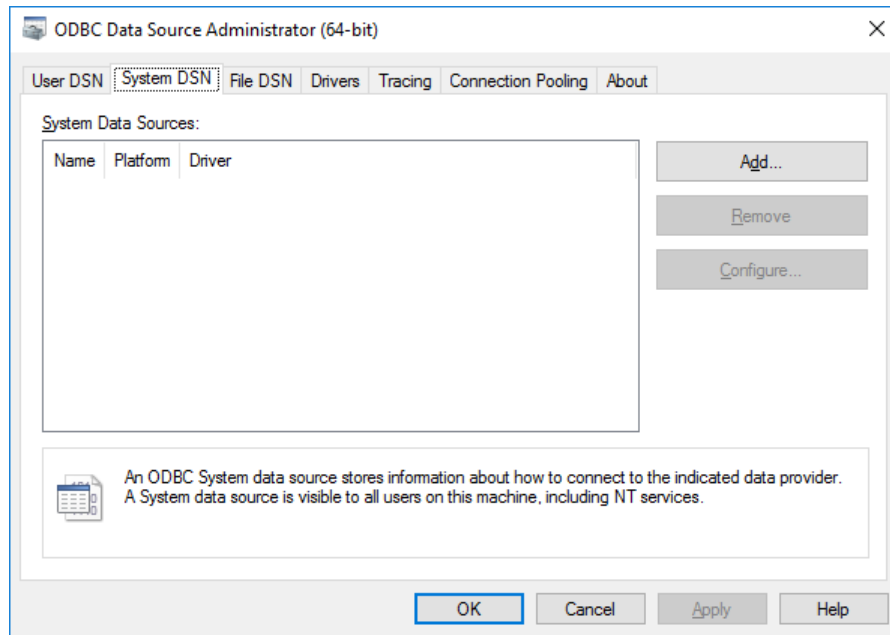
- 2 Select **Configure Data Source > Configure ODBC data source**.



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.

The screenshot shows the 'PostgreSQL ANSI ODBC Driver (psqlODBC) Setup' dialog box. It features a title bar with a close button (X). The main area contains the following fields and controls:

- Data Source:** A text input field.
- Description:** A text input field.
- Database:** A text input field.
- Server:** A text input field.
- User Name:** A text input field.
- Port:** A text input field with the value '5432'.
- Password:** A text input field.
- SSL Mode:** A dropdown menu with 'disable' selected.
- Options:** A section containing two buttons: 'Datasource' and 'Global'.
- Test:** A button located to the right of the 'Options' section.
- Save:** A button at the bottom center.
- Cancel:** A button at the bottom right.

- 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is PostgreSQL, and two tabs named PostgreSQL and PostgreSQL1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

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

close | database

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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.

In this section...
“Step 1. Verify the driver installation.” on page 2-72
“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-72
“Step 3. Set up the data source using the Database Explorer app.” on page 2-73
“Step 4. Connect using the Database Explorer app or the command line.” on page 2-75

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. Add the JDBC driver to the MATLAB static Java class path.

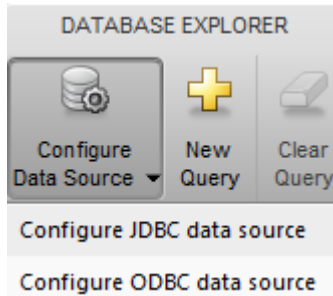
- 1 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `C:\DB_Drivers\postgresql-8.4-702.jdbc4.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to PostgreSQL Using JDBC Driver and Command Line” on page 2-76.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a data source name. (This example uses a data source named PostgreSQL.) You use this name to establish a connection to your database. If you enter a name that duplicates an existing ODBC data source name, the Database Explorer app appends `_JDBC` to the name after you save it.
- 4 From the **Vendor** list, select **POSTGRESQL**. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: `Unable to find JDBC driver file on MATLAB Java class path.` Address this message by following the steps described in Step 2 on page 2-72.

The screenshot shows a window titled "JDBC Data Source Configuration". It is divided into two main sections: "Data Source Details" and "Connection Parameters".

Data Source Details:

- Name:** A text box containing "PostgreSQL".
- Vendor:** A dropdown menu with the following options: MICROSOFT SQL SERVER, MYSQL, ORACLE, **POSTGRESQL** (highlighted), and OTHER.

Connection Parameters:

- Database:** An empty text box.
- Server:** A text box containing "localhost".
- Port Number:** A text box containing "5432".

At the bottom of the dialog, there are four buttons: "Edit", "Test", "Save", and "Delete". Below the buttons is a "Message" section, which is currently empty.

- 5 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.

- 6 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.

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

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

Step 4. 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source name you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is `Postgresql`, and two tabs named `Postgresql` and `Postgresql1` are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to PostgreSQL Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Use the `Vendor` name-value pair argument of the `database` function to specify a connection to a PostgreSQL database. 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` and the password `pwd`.

```
conn = database('dbname','username','pwd', ...  
              'Vendor','PostgreSQL', ...  
              'Server','sname');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`close` | `database` | `javaaddpath`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

SQLite JDBC for Windows

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

In this section...

“Step 1. Verify the driver installation.” on page 2-77

“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-77

“Step 3. Set up the data source using the Database Explorer app.” on page 2-78

“Step 4. Connect using the Database Explorer app or the command line.” on page 2-80

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 “Working with MATLAB Interface to SQLite” on page 2-6.

Step 2. Add the JDBC driver to the MATLAB static Java class path.

- 1 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`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `C:\DB_Drivers\sqlite-jdbc-3.7.2.jar`. Save and close `javaclasspath.txt`.

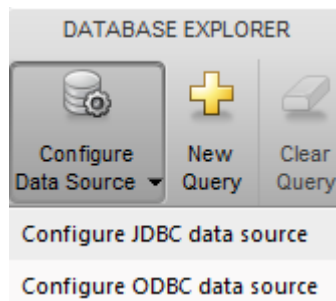
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

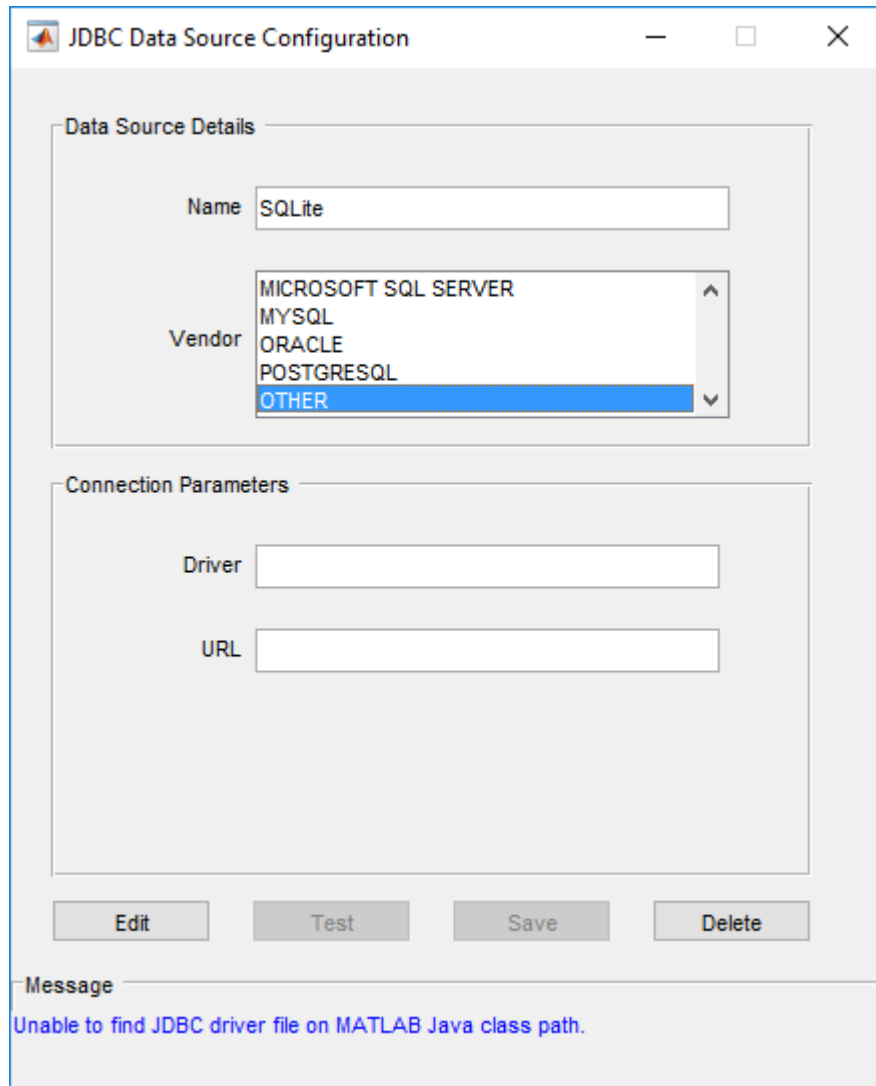
This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to SQLite Using JDBC Driver and Command Line” on page 2-81.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a data source name. (This example uses a data source named `SQLite`.) If you enter a name that duplicates an existing ODBC data source name, the Database Explorer app appends `_JDBC` to the name after you save it.
- 4 From the **Vendor** list, select **OTHER**.



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.

- 5 In the **Driver** box, enter the SQLite driver Java class object. Here, use `org.sqlite.JDBC`. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: **Unable to find JDBC driver file on MATLAB Java class path**. Address this message by following the steps described in Step 2 on page 2-77.
- 6 Connect to the SQLite database by creating a URL string using the format `jdbc:subprotocol:subname`. The `jdbc` part of this string stays constant for any JDBC driver. `subprotocol` is a database type, in this case, `sqlite`. The last part of the URL string is `subname`. For SQLite, `subname` contains the location of the database. For example, your 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**.
- 7 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.

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

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

Step 4. 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. Enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is SQLite, and two tabs named SQLite and SQLite1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to SQLite Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Connect to the SQLite database by using the `database` function.
 - Enter the data source name that you defined for the first argument.
 - Enter your user name `username` and password `pwd`, or leave these arguments blank if your database does not require them.
 - The fourth argument is the driver Java class object. This code assumes that the class object is `org.sqlite.JDBC`.
 - The last argument is the URL string `URL` that you create using the format `jdbc:subprotocol:subname`. The `jdbc` part of this string stays constant for any JDBC driver. `subprotocol` is a database type, in this case, `sqlite`. The last part of the URL string is `subname`. For SQLite, `subname` contains the location of the database. For example, your string is `jdbc:sqlite:dbpath`, where `dbpath` is the full path to your SQLite database on your computer.

For example, this code assumes that you are connecting to a data source named SQLite.

```
conn = database('SQLite', 'username', 'pwd', 'org.sqlite.JDBC', 'URL');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

close | database | javaaddpath

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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.

In this section...

“Step 1. Verify the driver installation.” on page 2-83

“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-83

“Step 3. Set up the data source using the Database Explorer app.” on page 2-84

“Step 4. Connect using the Database Explorer app or the command line.” on page 2-86

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. Add the JDBC driver to the MATLAB static Java class path.

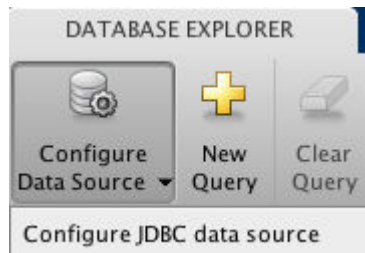
- 1 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `/home/user/DB_Drivers/sqljdbc4.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

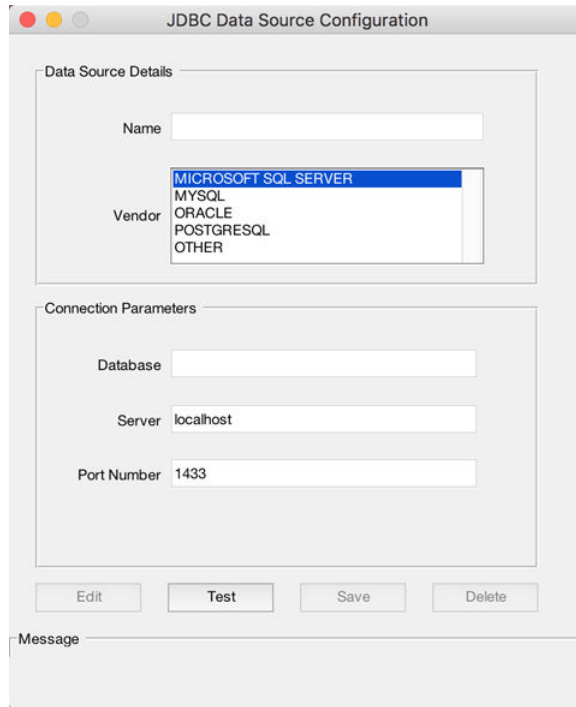
Step 3. Set up the data source using the Database Explorer app.

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to SQL Server Using JDBC Driver and Command Line” on page 2-86.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.



- 3 In the **Name** box, enter a data source name.
- 4 From the **Vendor** list, select MICROSOFT SQL SERVER. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: Unable to find JDBC driver file on MATLAB Java class path. Address this message by following the steps described in Step 2 on page 2-83.
- 5 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.
- 6 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.

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

Step 4. 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is MS SQL Server, and two tabs named MS SQL Server and MS SQL Server1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to SQL Server Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Use the Vendor name-value pair argument of the `database` function to specify a connection to a SQL Server 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);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`close` | `database` | `javaaddpath`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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.

In this section...
“Step 1. Verify the driver installation.” on page 2-88
“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-88
“Step 3. Set up the data source using the Database Explorer app.” on page 2-89
“Step 4. Connect using the Database Explorer app or the command line.” on page 2-91

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. Add the JDBC driver to the MATLAB static Java class path.

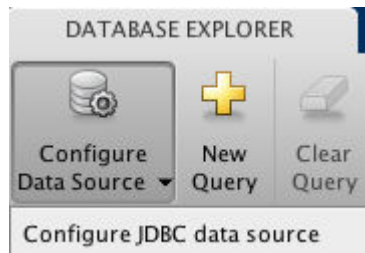
- 1 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `/home/user/DB_Drivers/sqljdbc4.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

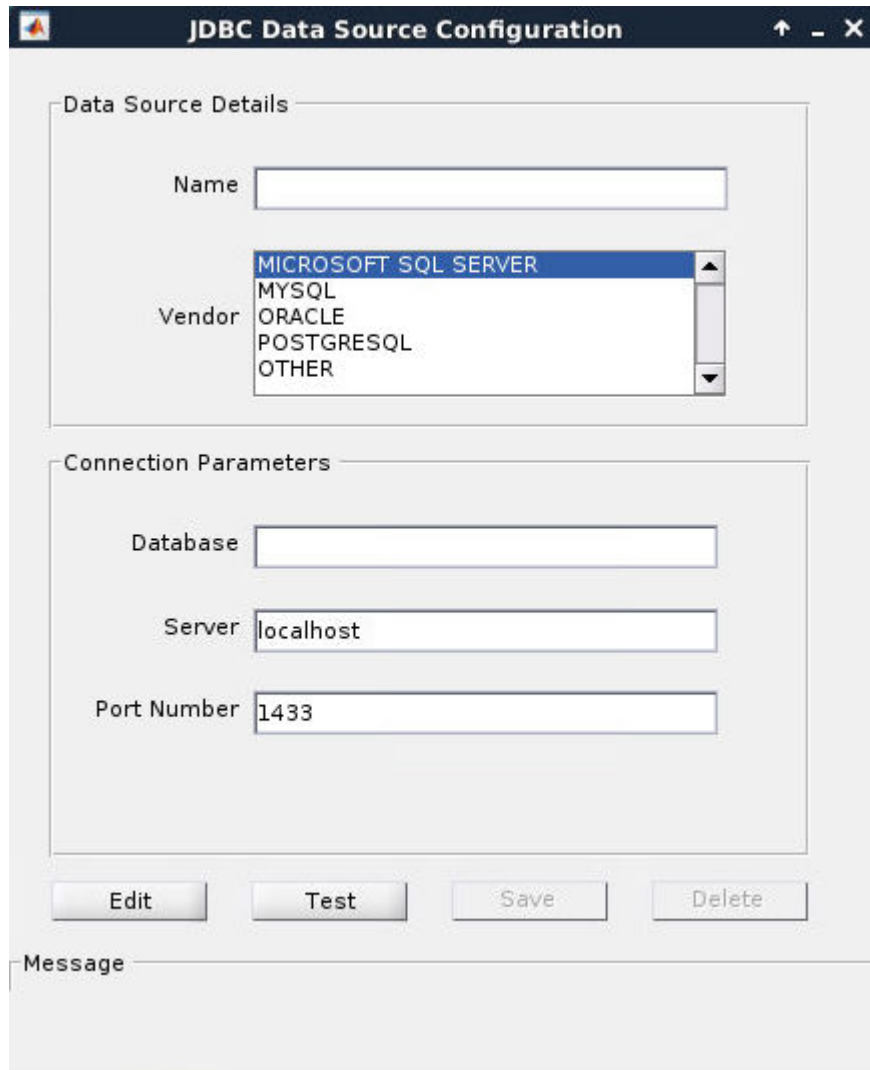
Step 3. Set up the data source using the Database Explorer app.

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to SQL Server Using JDBC Driver and Command Line” on page 2-92.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.



- 3 In the **Name** box, enter a data source name.
- 4 From the **Vendor** list, select MICROSOFT SQL SERVER. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: Unable to find JDBC driver file on MATLAB Java class path. Address this message by following the steps described in Step 2 on page 2-88.

- 5 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. From the **Authentication** list, select **Server**.
- 6 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.

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

Step 4. 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is

MS SQL Server, and two tabs named MS SQL Server and MS SQL Server1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to SQL Server Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Use the `Vendor` name-value pair argument of the `database` function to specify a connection to a SQL Server 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);
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`close` | `database` | `javaaddpath`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14

- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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.

In this section...

“Step 1. Verify the driver installation.” on page 2-94

“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-94

“Step 3. Set up the data source using the Database Explorer app.” on page 2-95

“Step 4. Connect using the Database Explorer app or the command line.” on page 2-97

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. Add the JDBC driver to the MATLAB static Java class path.

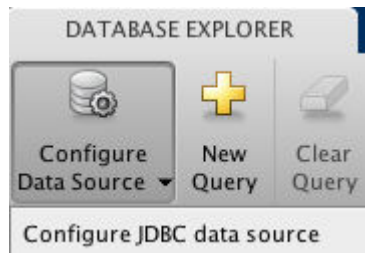
- 1 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `/home/user/DB_Drivers/ojdbc6.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to Oracle Using JDBC Driver and Command Line” on page 2-98.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a data source name. (This example uses a data source named `ORA`.)
- 4 From the **Vendor** list, select `ORACLE`. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: `Unable to find JDBC driver file on MATLAB Java class path`. Address this message by following the steps described in Step 2 on page 2-94.

JDBC Data Source Configuration

Data Source Details

Name ORA

Vendor
MICROSOFT SQL SERVER
MYSQL
ORACLE
POSTGRESQL
OTHER

Connection Parameters

Database

Server localhost

Port Number 1521

Driver Type thin

Edit Test Save Delete

Message

- 5 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. From the **Driver Type** list, select `thin` or `oci`. (Use `thin` as the default driver. Use `oci` if you installed an OCI driver.)
- 6 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.

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

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

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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is ORA, and two tabs named ORA and ORA1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to Oracle Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Use the `Vendor` name-value pair argument of the `database` function to specify a connection to an Oracle database. Set the `DriverType` name-value pair argument to `thin`. 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','Oracle','DriverType','thin', ...  
              'Server','sname','PortNumber',123456);
```

`dbname` 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 in `<ORACLE_HOME>/NETWORK/ADMIN` where `<ORACLE_HOME>` is the folder containing the installed database or the Oracle client.

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

close | database | javaaddpath

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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.

In this section...

“Step 1. Verify the driver installation.” on page 2-100

“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-100

“Step 3. Set up the data source using the Database Explorer app.” on page 2-101

“Step 4. Connect using the Database Explorer app or the command line.” on page 2-103
--

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. Add the JDBC driver to the MATLAB static Java class path.

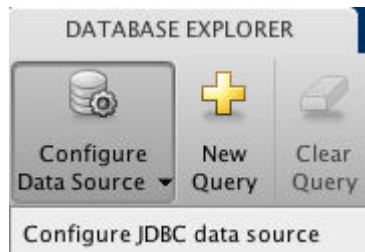
- 1 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `/home/user/DB_Drivers/ojdbc6.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to Oracle Using JDBC Driver and Command Line” on page 2-104.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a data source name. (This example uses a data source named `ORA`.)
- 4 From the **Vendor** list, select `ORACLE`. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: `Unable to find JDBC driver file on MATLAB Java class path`. Address this message by following the steps described in Step 2 on page 2-100.

JDBC Data Source Configuration

Data Source Details

Name: ORA

Vendor: MICROSOFT SQL SERVER, MYSQL, ORACLE, POSTGRESQL, OTHER

Connection Parameters

Database:

Server: localhost

Port Number: 1521

Driver Type: thin

Edit Test Save Delete

Message

- 5 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. From the **Driver Type** list, select `thin` or `oci`. (Use `thin` as the default driver. Use `oci` if you installed an OCI driver.)

- 6 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.

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

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

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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is ORA, and two tabs named ORA and ORA1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to Oracle Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Use the `Vendor` name-value pair argument of the `database` function to specify a connection to an Oracle database. Set the `DriverType` name-value pair argument to `thin`. 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','Oracle','DriverType','thin', ...  
              'Server','sname','PortNumber',123456);
```

`dbname` 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 in `<ORACLE_HOME>/NETWORK/ADMIN` where `<ORACLE_HOME>` is the folder containing the installed database or the Oracle client.

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

close | database | javaaddpath

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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 5.1.17 driver to connect to a MySQL Version 5.5.16 database.

In this section...
“Step 1. Verify the driver installation.” on page 2-106
“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-106
“Step 3. Set up the data source using the Database Explorer app.” on page 2-107
“Step 4. Connect using the Database Explorer app or the command line.” on page 2-109

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. Add the JDBC driver to the MATLAB static Java class path.

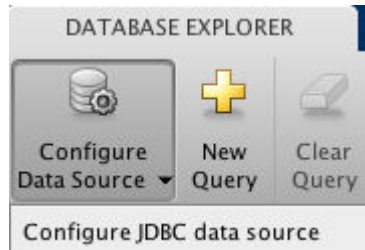
- 1 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `/home/user/DB_Drivers/mysql-connector-java-5.1.17-bin.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to MySQL Using JDBC Driver and Command Line” on page 2-110.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a data source name. (This example uses a data source named MySQL.)
- 4 From the **Vendor** list, select **MYSQL**. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: **Unable to find JDBC driver file on MATLAB Java class path**. Address this message by following the steps described in Step 2 on page 2-106.

JDBC Data Source Configuration

Data Source Details

Name MySQL

Vendor

- MICROSOFT SQL SERVER
- MYSQL**
- ORACLE
- POSTGRESQL
- OTHER

Connection Parameters

Database

Server localhost

Port Number 3306

Edit Test Save Delete

Message

- 5 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.
- 6 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.

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

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

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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is MySQL, and two tabs named MySQL and MySQL1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to MySQL Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Use the Vendor name-value pair argument of the `database` function to specify a connection to a MySQL database. 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` and the password `pwd`.

```
conn = database('dbname','username','pwd', ...  
              'Vendor','MySQL', ...  
              'Server','sname');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`close` | `database` | `javaaddpath`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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 5.1.17 driver to connect to a MySQL Version 5.5.16 database.

In this section...

“Step 1. Verify the driver installation.” on page 2-111

“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-111

“Step 3. Set up the data source using the Database Explorer app.” on page 2-112

“Step 4. Connect using the Database Explorer app or the command line.” on page 2-114

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. Add the JDBC driver to the MATLAB static Java class path.

- 1 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`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `/home/user/DB_Drivers/mysql-connector-java-5.1.17-bin.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to MySQL Using JDBC Driver and Command Line” on page 2-115.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a data source name. (This example uses a data source named MySQL.)
- 4 From the **Vendor** list, select **MYSQL**. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: **Unable to find JDBC driver file on MATLAB Java class path**. Address this message by following the steps described in Step 2 on page 2-111.

JDBC Data Source Configuration

Data Source Details

Name: MySQL

Vendor: MICROSOFT SQL SERVER, **MYSQL**, ORACLE, POSTGRESQL, OTHER

Connection Parameters

Database:

Server: localhost

Port Number: 3306

Edit Test Save Delete

Message

- 5 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.

- 6 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.

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

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

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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is MySQL, and two tabs named MySQL and MySQL1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to MySQL Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Use the Vendor name-value pair argument of the `database` function to specify a connection to a MySQL database. 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` and the password `pwd`.

```
conn = database('dbname','username','pwd', ...  
              'Vendor','MySQL', ...  
              'Server','sname');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`close` | `database` | `javaaddpath`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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.

In this section...
“Step 1. Verify the driver installation.” on page 2-116
“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-116
“Step 3. Set up the data source using the Database Explorer app.” on page 2-117
“Step 4. Connect using the Database Explorer app or the command line.” on page 2-119

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. Add the JDBC driver to the MATLAB static Java class path.

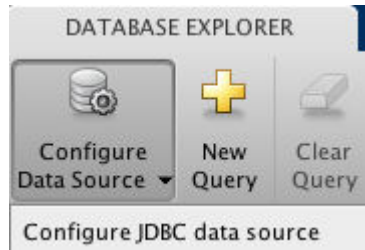
- 1 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `/home/user/DB_Drivers/postgresql-8.4-702.jdbc4.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to PostgreSQL Using JDBC Driver and Command Line” on page 2-120.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a data source name. (This example uses a data source named PostgreSQL.)
- 4 From the **Vendor** list, select **POSTGRESQL**. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom:
Unable to find JDBC driver file on MATLAB Java class path. Address this message by following the steps described in Step 2 on page 2-116.

JDBC Data Source Configuration

Data Source Details

Name PostgreSQL

Vendor

- MICROSOFT SQL SERVER
- MYSQL
- ORACLE
- POSTGRESQL
- OTHER

Connection Parameters

Database

Server localhost

Port Number 5432

Edit Test Save Delete

Message

- 5 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.
- 6 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.

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

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

Step 4. 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is

PostgreSQL, and two tabs named PostgreSQL and PostgreSQL1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to PostgreSQL Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Use the Vendor name-value pair argument of the `database` function to specify a connection to a PostgreSQL database. 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` and the password `pwd`.

```
conn = database('dbname', 'username', 'pwd', ...  
              'Vendor', 'PostgreSQL', ...  
              'Server', 'sname');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`close` | `database` | `javaaddpath`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14

- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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. This tutorial uses the JDBC4 PostgreSQL Driver, Version 8.4 to connect to a PostgreSQL 9.2 database.

In this section...
“Step 1. Verify the driver installation.” on page 2-122
“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-122
“Step 3. Set up the data source using the Database Explorer app.” on page 2-123
“Step 4. Connect using the Database Explorer app or the command line.” on page 2-125

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. Add the JDBC driver to the MATLAB static Java class path.

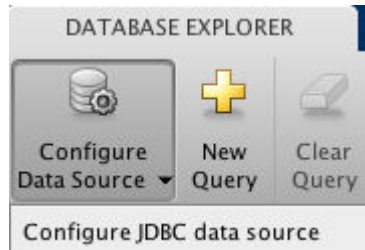
- 1 Run the `prefdir` function in the Command Window. The output of this command is a file path to the MATLAB preferences folder on your computer. For details, see `prefdir`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `/home/user/DB_Drivers/postgresql-8.4-702.jdbc4.jar`. Save and close `javaclasspath.txt`.
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

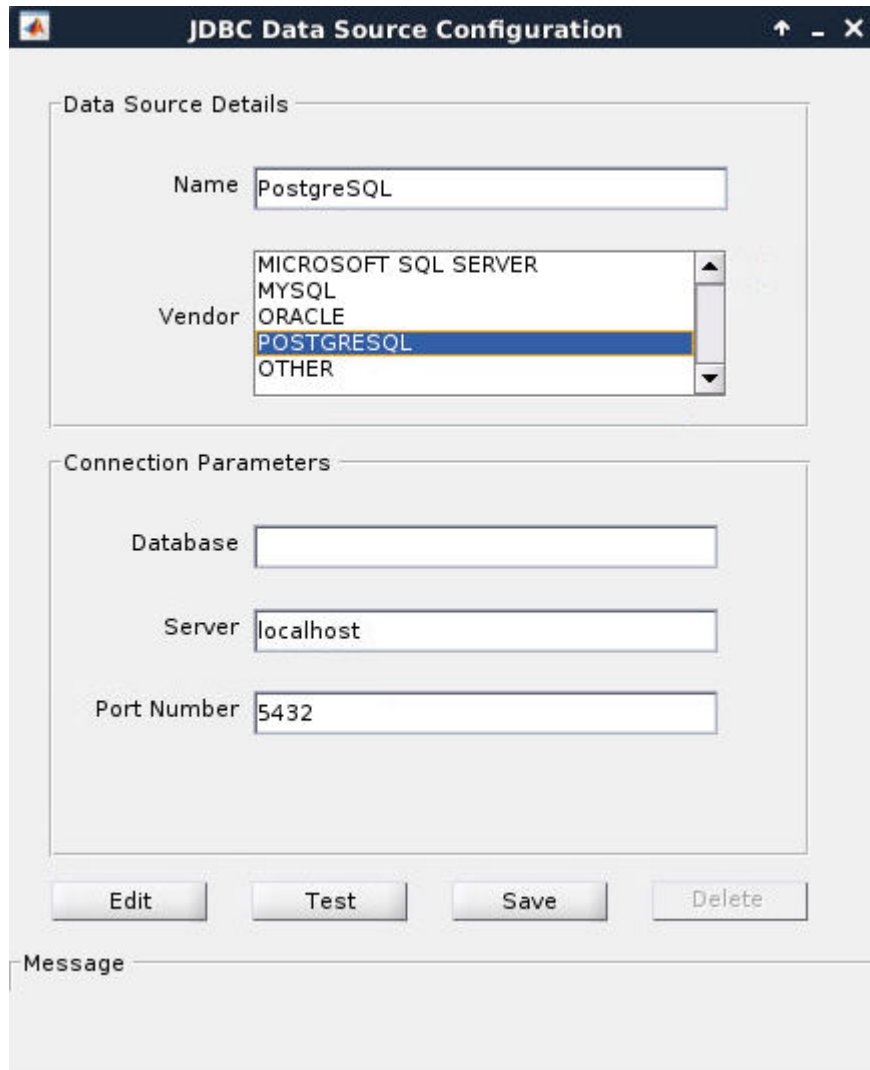
This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to PostgreSQL Using JDBC Driver and Command Line” on page 2-126.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a data source name. (This example uses a data source named PostgreSQL.)
- 4 From the **Vendor** list, select `POSTGRESQL`. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: `Unable to find JDBC driver file on MATLAB Java class path. Address this message by following the steps described in Step 2 on page 2-122.`



- 5 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.

- 6 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.

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

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

Step 4. 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. 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 Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is PostgreSQL, and two tabs named PostgreSQL and PostgreSQL1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to PostgreSQL Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Use the `Vendor` name-value pair argument of the `database` function to specify a connection to a PostgreSQL database. 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` and the password `pwd`.

```
conn = database('dbname','username','pwd', ...  
              'Vendor','PostgreSQL', ...  
              'Server','sname');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

`close` | `database` | `javaaddpath`

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

SQLite JDBC for macOS

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

In this section...

“Step 1. Verify the driver installation.” on page 2-127

“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-127

“Step 3. Set up the data source using the Database Explorer app.” on page 2-128

“Step 4. Connect using the Database Explorer app or the command line.” on page 2-130

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 “Working with MATLAB Interface to SQLite” on page 2-6.

Step 2. Add the JDBC driver to the MATLAB static Java class path.

- 1 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`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `/home/user/DB_Drivers/sqlite-jdbc-3.7.2.jar`. Save and close `javaclasspath.txt`.

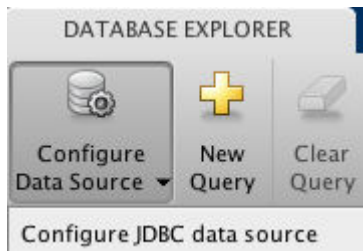
- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

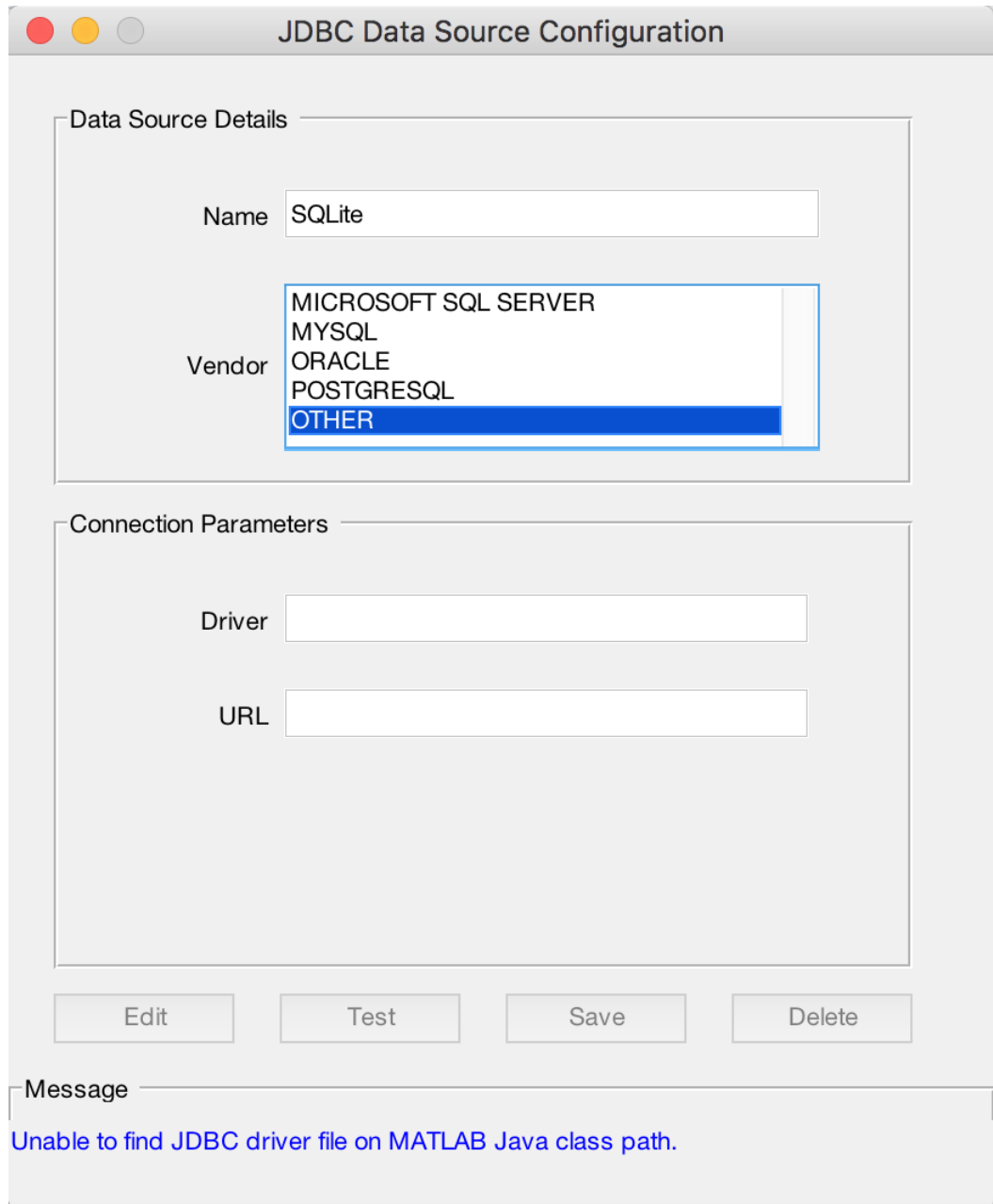
This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to SQLite Using JDBC Driver and Command Line” on page 2-131.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a data source name. (This example uses a data source named SQLite.)
- 4 From the **Vendor** list, select OTHER.



JDBC Data Source Configuration

Data Source Details

Name

Vendor

Connection Parameters

Driver

URL

Message

Unable to find JDBC driver file on MATLAB Java class path.

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.

- 5 In the **Driver** box, enter the SQLite driver Java class object. Here, use `org.sqlite.JDBC`. If you did not add the JDBC driver file path to the Java class path, the dialog box displays this message at the bottom: `Unable to find JDBC driver file on MATLAB Java class path`. Address this message by following the steps described in Step 2 on page 2-127.
- 6 Connect to the SQLite database by creating a URL string using the format `jdbc:subprotocol:subname`. The `jdbc` part of this string stays constant for any JDBC driver. `subprotocol` is a database type, in this case, `sqlite`. The last part of the URL string is `subname`. For SQLite, `subname` contains the location of the database. For example, your 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**.
- 7 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.

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

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

Step 4. 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. Enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is SQLite, and two tabs named SQLite and SQLite1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to SQLite Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Connect to the SQLite database by using the database function.
 - Enter the data source name that you defined for the first argument.
 - Enter your user name `username` and password `pwd`, or leave these arguments blank if your database does not require them.
 - The fourth argument is the driver Java class object. This code assumes that the class object is `org.sqlite.JDBC`.
 - The last argument is the URL string `URL` that you create using the format `jdbc:subprotocol:subname`. The `jdbc` part of this string stays constant for any JDBC driver. `subprotocol` is a database type, in this case, `sqlite`. The last part of the URL string is `subname`. For SQLite, `subname` contains the location of the database. For example, your string is `jdbc:sqlite:dbpath`, where `dbpath` is the full path to your SQLite database on your computer.

For example, this code assumes that you are connecting to a data source named SQLite.

```
conn = database('SQLite', 'username', 'pwd', 'org.sqlite.JDBC', 'URL');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps

Database Explorer

Functions

close | database | javaaddpath

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

SQLite JDBC for Linux

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

In this section...

“Step 1. Verify the driver installation.” on page 2-133

“Step 2. Add the JDBC driver to the MATLAB static Java class path.” on page 2-133

“Step 3. Set up the data source using the Database Explorer app.” on page 2-134

“Step 4. Connect using the Database Explorer app or the command line.” on page 2-136

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 “Working with MATLAB Interface to SQLite” on page 2-6.

Step 2. Add the JDBC driver to the MATLAB static Java class path.

- 1 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`.
- 2 Close MATLAB.
- 3 Navigate to the folder from step 1, and create a file named `javaclasspath.txt` in the folder.
- 4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider followed by the JAR file name. The following is an example of the path: `/home/user/DB_Drivers/sqlite-jdbc-3.7.2.jar`. Save and close `javaclasspath.txt`.

- 5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add a JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).

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

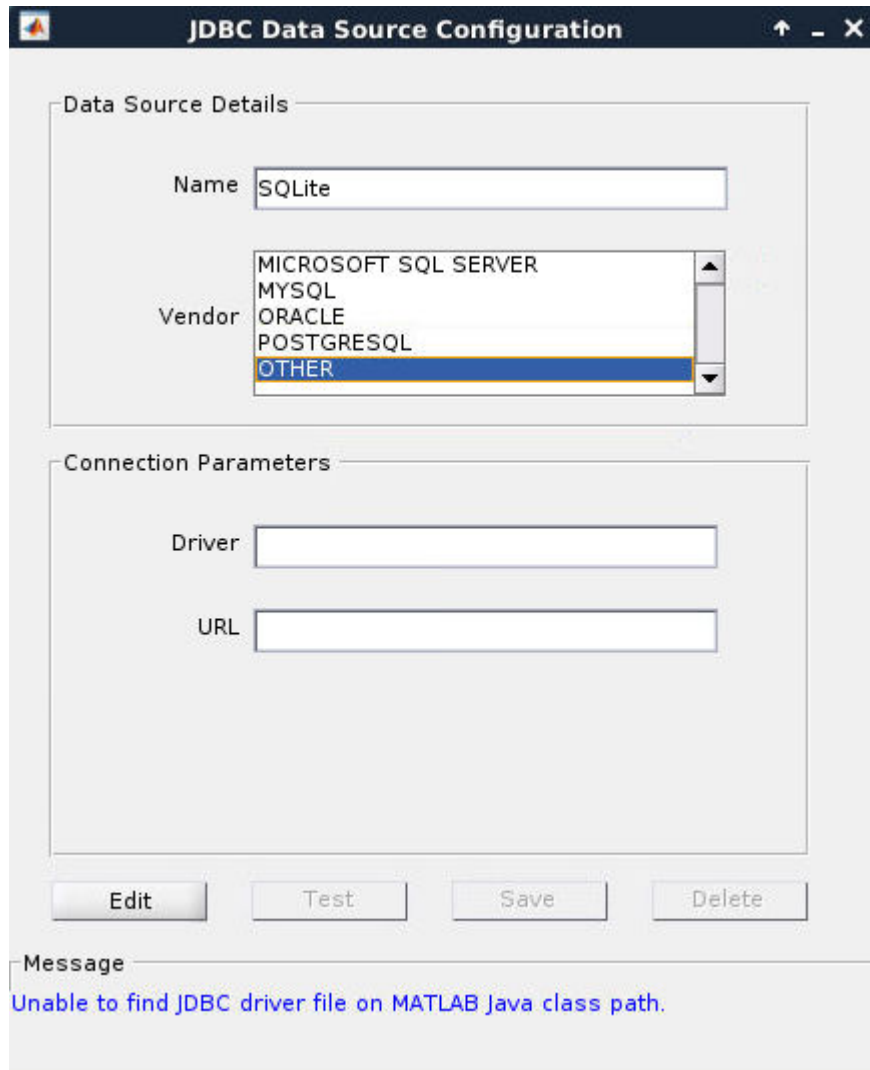
This step is required only for connecting to a database using the Database Explorer app. If you want to use the command line to connect to your database, see “Connect to SQLite Using JDBC Driver and Command Line” on page 2-137.

- 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 Select **Configure Data Source > Configure JDBC data source**.



The JDBC Data Source Configuration dialog box opens.

- 3 In the **Name** box, enter a data source name. (This example uses a data source named SQLite.)
- 4 From the **Vendor** list, select OTHER.



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.

- 5 In the **Driver** box, enter the SQLite driver Java class object. Here, use `org.sqlite.JDBC`. If you did not add the JDBC driver file path to the Java class

path, the dialog box displays this message at the bottom: **Unable to find JDBC driver file on MATLAB Java class path.** Address this message by following the steps described in Step 2 on page 2-133.

- 6 Connect to the SQLite database by creating a URL string using the format `jdbc:subprotocol:subname`. The `jdbc` part of this string stays constant for any JDBC driver. `subprotocol` is a database type, in this case, `sqlite`. The last part of the URL string is `subname`. For SQLite, `subname` contains the location of the database. For example, your 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**.
- 7 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.

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

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

Step 4. 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 **Data Source** section, click **New Query**.
- 2 In the Connect to a Data Source dialog box, select the data source you defined from the **Data Source** list. Enter a user name and password, or leave these boxes blank if your database does not require them. Click **Connect**.

The Database Explorer app connects to the database and displays database 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 and the database connection.

Tip To close the database connection, close all tabs that have titles beginning with the name of the corresponding data source. For example, if the data source name is SQLite, and two tabs named SQLite and SQLite1 are open, then close both tabs.

To close all database connections, close the Database Explorer app.

Connect to SQLite Using JDBC Driver and Command Line

When using the command line, you do not have to set up a data source with the Database Explorer app. You can use the command line to pass all the required parameters for connection.

- 1 Connect to the SQLite database by using the `database` function.
 - Enter the data source name that you defined for the first argument.
 - Enter your user name `username` and password `pwd`, or leave these arguments blank if your database does not require them.
 - The fourth argument is the driver Java class object. This code assumes that the class object is `org.sqlite.JDBC`.
 - The last argument is the URL string `URL` that you create using the format `jdbc:subprotocol:subname`. The `jdbc` part of this string stays constant for any JDBC driver. `subprotocol` is a database type, in this case, `sqlite`. The last part of the URL string is `subname`. For SQLite, `subname` contains the location of the database. For example, your string is `jdbc:sqlite:dbpath`, where `dbpath` is the full path to your SQLite database on your computer.

For example, this code assumes that you are connecting to a data source named SQLite.

```
conn = database('SQLite', 'username', 'pwd', 'org.sqlite.JDBC', 'URL');
```

- 2 Close the database connection.

```
close(conn)
```

See Also

Apps
Database Explorer

Functions

close | database | javaaddpath

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

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 “Configuring Driver and Data Source” on page 2-15. You can create a data source. Then, 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 with MATLAB. 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.
- 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 2-55.

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

These steps show how to configure a driver and connect to a JDBC-compliant database with MATLAB. 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.

- 1 If your driver is not preinstalled on your computer, find a compatible driver and install it on your computer.
- 2 Add the JDBC driver path to the static Java class path or the dynamic Java class path. For details about static and dynamic class paths, see “Java Class Path” (MATLAB).
- 3 To connect to a JDBC-compliant database, you must know your database driver Java class object. For example, the Java class object for a SQLite database driver is `org.sqlite.JDBC`. Specify this driver value using either the **Driver** box in the JDBC Data Source Configuration dialog box or the `driver` input argument of the `database` function at the command line.

Note For JDBC-compliant databases, specify the driver and URL in the JDBC Data Source Configuration dialog box of the Database Explorer app or in the `database` function. The driver and the URL string can vary depending on the type and version of the JDBC driver and the database you are working with. For details about the driver and URL, see the JDBC driver documentation for your database.

- 4 To connect to a JDBC-compliant database, you must create a URL string. The URL string is in the form `jdbc:subprotocol:subname`. The `jdbc` part of this string stays constant for any JDBC driver. `subprotocol` is the database type. The last part of the URL string is `subname`. 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 string is `jdbc:sqlite:dbpath`, where `dbpath` is the full path to your SQLite database on your computer. Use this string for establishing a connection either with the Database Explorer app or the command line.
- 5 Use the Database Explorer app to test your connection and connect to your database. For an example, see “SQLite JDBC for Windows” on page 2-77.

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

See Also

Apps

Database Explorer

Functions

`close` | `database` | `javaaddpath`

Related Examples

- “MySQL ODBC for Windows” on page 2-55
- “SQLite JDBC for Windows” on page 2-77

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Java Class Path” (MATLAB)
- “Modify and Delete Data Sources” on page 4-14
- “Database Explorer App Error Messages” on page 3-16
- “Database Connection Error Messages” on page 3-9

Connecting to Database

To connect to a database from MATLAB, install an ODBC or JDBC driver and create a data source. For details about driver installation and data source setup, see “Configuring Driver and Data Source” on page 2-15. 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 “Working with MATLAB Interface to SQLite” on page 2-6.

You can connect to a database using the **Database Explorer** 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 2-9.

After deciding between the Database Explorer app and the command line, connect to your database by following the steps for the corresponding database and driver type, as given in this list.

Microsoft Access

- ODBC
 - “Connect to Access Using Database Explorer App” on page 2-21
 - “Connect to Access Using ODBC Driver and Command Line” on page 2-22

Microsoft SQL Server

- ODBC
 - “Connect to SQL Server Using Database Explorer App” on page 2-29
 - “Connect to SQL Server Using ODBC Driver and Command Line” on page 2-30
- JDBC
 - “Connect to SQL Server Using Database Explorer App” on page 2-39
 - “Connect to SQL Server Using JDBC Driver and Command Line” on page 2-40

Oracle

- ODBC

- “Connect to Oracle Using Database Explorer App” on page 2-46
- “Connect to Oracle Using ODBC Driver and Command Line” on page 2-46
- JDBC
 - “Connect to Oracle Using Database Explorer App” on page 2-52
 - “Connect to Oracle Using JDBC Driver and Command Line” on page 2-53

MySQL

- ODBC
 - “Connect to MySQL Using Database Explorer App” on page 2-59
 - “Connect to MySQL Using ODBC Driver and Command Line” on page 2-59
- JDBC
 - “Connect to MySQL Using Database Explorer App” on page 2-64
 - “Connect to MySQL Using JDBC Driver and Command Line” on page 2-65

PostgreSQL

- ODBC
 - “Connect to PostgreSQL Using Database Explorer App” on page 2-70
 - “Connect to PostgreSQL Using ODBC Driver and Command Line” on page 2-70
- JDBC
 - “Connect to PostgreSQL Using Database Explorer App” on page 2-75
 - “Connect to PostgreSQL Using JDBC Driver and Command Line” on page 2-76

SQLite

- JDBC
 - “Connect to SQLite Using Database Explorer App” on page 2-80
 - “Connect to SQLite Using JDBC Driver and Command Line” on page 2-81

- No driver or database installation is required. To create a SQLite connection, use `sqlite` to create a new SQLite database file or connect to an existing SQLite database file.

Other ODBC-Compliant or JDBC-Compliant Databases

For an example of how to connect to a database that is not listed here, see “Other ODBC-Compliant or JDBC-Compliant Databases” on page 2-139.

See Also

`close | database`

More About

- “Setup Requirements for Database Connection” on page 2-12
- “Access Relational Database Data in MATLAB” on page 2-3
- “Connection Options” on page 2-9
- “Choosing Between ODBC and JDBC Drivers” on page 2-13
- “Configuring Driver and Data Source” on page 2-15
- “Working with MATLAB Interface to SQLite” on page 2-6

Data Import Using Database Explorer App or Command Line

In this section...

“Data Import Using Database Explorer App” on page 2-145

“Data Import Using Command Line” on page 2-146

“Custom Data Types” on page 2-147

“SQL Queries Saved in Scripts or Files” on page 2-147

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 “Working with MATLAB Interface to SQLite” on page 2-6.

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 4-17. You can embed the generated SQL code into the SQL query that you specify in the `exec` function. Or, you can create an SQL script file to use with the `runsqlscript` 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 4-18.

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 `sql read` function at the command line. This function needs only a database connection and the database table name to import data. Furthermore, the `sql read` 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 Databases into MATLAB” on page 5-3. If you have variables in the MATLAB workspace, you can add them to the SQL query. For an example, see “Create Queries with Characters and Variables” on page 5-7.

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 in one step. Use the `exec` and `fetch` functions for a two-step approach with maximum flexibility for setting database preferences and importing numeric data with double precision. The `exec` function executes the SQL statement, and the `fetch` function imports the data from the database into a MATLAB variable. This table provides more details about these ways of importing data into MATLAB.

Functionality	One-Step Data Import	Two-Step Data Import
Function	<code>select</code>	<code>exec</code> and <code>fetch</code>
SQL statement	Single SELECT statement only	Single or multiple SELECT statements
Data types for imported numeric values	Specified by the database table definition	MATLAB double
Memory management	Integer classes for numeric values	Number of imported rows or database preferences
Database preferences	Setting database preferences not required	Setting database preferences required using <code>setdbprefs</code>

For memory management, see “Data Import Approaches and Memory Management” on page 5-46.

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

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 `runsqlscript` function. If you have SQL queries stored in `.sql` or text files that you want to run from MATLAB, you also can use this function.

See Also

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

More About

- “Connection Options” on page 2-9
- “Connecting to Database Using Native ODBC Interface” on page 3-19
- “Data Import Approaches and Memory Management” on page 5-46
- “Working with Large Data Sets” on page 2-150
- “Working with MATLAB Interface to SQLite” on page 2-6
- “Data Type Support” on page 1-3
- “Data Retrieval Restrictions” on page 1-5

Inserting Data Using Command Line

You can insert data from MATLAB into a database using the command line. The `sqlwrite` function inserts data from the MATLAB workspace into a database.

Note The `datainsert` and `fastinsert` functions will be removed in a future release. Also, the syntax of the `insert` function will not support the connection object in a future release. Use the `sqlwrite` function instead.

The MATLAB interface to SQLite supports the `insert` function only. For details about this interface, see “Working with MATLAB Interface to SQLite” on page 2-6.

The `sqlwrite` function:

- Inserts data into an existing database table
- Creates a database table if the specified table name does not exist in the database, and inserts data into the new table
- Inserts data specified as a `datetime` array, `string`, or specific integer class
- Does not require preprocessing and conversion of data to specific formats
- Does not require SQL knowledge
- Does not require you to set database preferences

The three functions `datainsert`, `fastinsert`, and `insert` behave differently depending on the database connection type.

- For ODBC connections, all three functions have identical functionality.
- For JDBC connections:
 - The `datainsert` function has faster performance.
 - Both the `fastinsert` and `insert` functions have identical functionality.

These three functions require special formatting of the insert data for dates and timestamps. The `datainsert` function also requires special formatting for `null` and `NaN` values.

All database preference settings, except `NullNumberWrite` and `NullStringWrite`, apply to the three functions. These two settings do not apply to the `datainsert` function. To set database preferences, use the `setdbprefs` function.

If you experience performance issues using these three functions, then use the bulk insert functionality of your database. For details, see “Export Data Using Bulk Insert” on page 5-32.

To learn about migrating from using these three functions to using the `sqlwrite` function, see “Append Data to Existing Database Table Using Insert Functionality” on page 5-80 and “Insert Data into New Database Table Using Insert Functionality” on page 5-82.

To import data into MATLAB from your database, use the `sqlread`, `select`, or `exec` and `fetch` functions.

See Also

database | update

More About

- “Export Data to New Record in Database” on page 5-21
- “Export Multiple Records from MATLAB Workspace” on page 5-27
- “Export Data Using Bulk Insert” on page 5-32
- “Import Data from Databases into MATLAB” on page 5-3
- “Import Data Using MATLAB® Interface to SQLite” on page 5-68
- “Writing Data Common Errors” on page 3-2
- “Insert Data into Database Table” on page 5-88
- “Append Data to Existing Database Table Using Insert Functionality” on page 5-80
- “Insert Data into New Database Table Using Insert Functionality” on page 5-82

Working with Large Data Sets

In this section...
“Connect to a Database with Maximum Performance” on page 2-150
“Import Large Data Sets into MATLAB” on page 2-150
“Export Large Data Sets from MATLAB” on page 2-151
“Access Large Data Using a DatabaseDatastore” on page 2-151

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. For details, see “Connecting to Database Using Native ODBC Interface” on page 3-19. If the native ODBC interface does not work, connect to your database using a JDBC driver. For details, see “Connecting to Database” on page 2-142.

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 `row limit` argument. Using a MATLAB script, you can fetch data in increments using the `row limit` 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 “Working with MATLAB Interface to SQLite” on page 2-6.

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 `datainsert` 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 `insert` with the MATLAB interface to SQLite. For details, see “Working with MATLAB Interface to SQLite” on page 2-6.

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”. Or, for more control, you can also write your own algorithms using `MapReduce`. For details, see “Analyze Large Data in Database Using MapReduce”.

See Also

More About

- “Choosing Between ODBC and JDBC Drivers” on page 2-13
- “Data Import Using Database Explorer App or Command Line” on page 2-145
- “Import Data from Databases into MATLAB” on page 5-3
- “Export Data to New Record in Database” on page 5-21

Deploying Database Application with MATLAB Compiler

In this section...
“Create and Deploy Database Application” on page 2-152
“About Driver Configurations” on page 2-152

If you want to share your MATLAB code with others in your organization, then you must create a standalone database application using MATLAB Compiler™. After compiling the database application, you can deploy it to the target machines. Use this procedure and driver-specific information to create and deploy a database application.

Create and Deploy Database Application

- 1 Write your database application code and save it as a MATLAB function in a file. Do not save the code as a MATLAB script file. Write the code in function form for database application deployment. Further, you must keep certain things in mind as you write your database application code. For details, see “Write Deployable MATLAB Code” (MATLAB Compiler).
- 2 Compile your database application with MATLAB Compiler using the standalone application packaging process. For details, see “Create Standalone Application from MATLAB” (MATLAB Compiler).
- 3 The generated output from the compilation process contains a folder called `for_testing`. Conduct a test on a target machine using the files found in this folder.
- 4 After the test is successful, you can distribute the database application to the target machines in your organization.

About Driver Configurations

Ensure the target machines have the correct driver configuration for your database application. See the following driver-specific tasks to configure data sources and drivers.

Native ODBC and ODBC Drivers

After compiling your database application, you must define the data sources referenced in your code on the target machine using the ODBC Data Source Administrator. Then, you can run your database application on the target machine.

JDBC Drivers

For applications that use JDBC drivers, use either option to specify the JDBC driver on the target machine:

- Use `javaaddpath` in your function code to add your JDBC driver JAR file. Do not include the JAR file in the `javaclasspath.txt` file.
- Add the JDBC driver JAR file to your `javaclasspath.txt` file. Do not use `javaaddpath` in your function code. For Microsoft SQL Server operating system authentication, add the full path of the library file to the `javalibrarypath.txt` file. For details, see “Microsoft SQL Server JDBC for Windows” on page 2-32.

Caution: Do not add driver JAR files using `javaclasspath` as this might cause issues on the target machine. For details, see “Java Class Path” (MATLAB).

See Also

`javaaddpath`

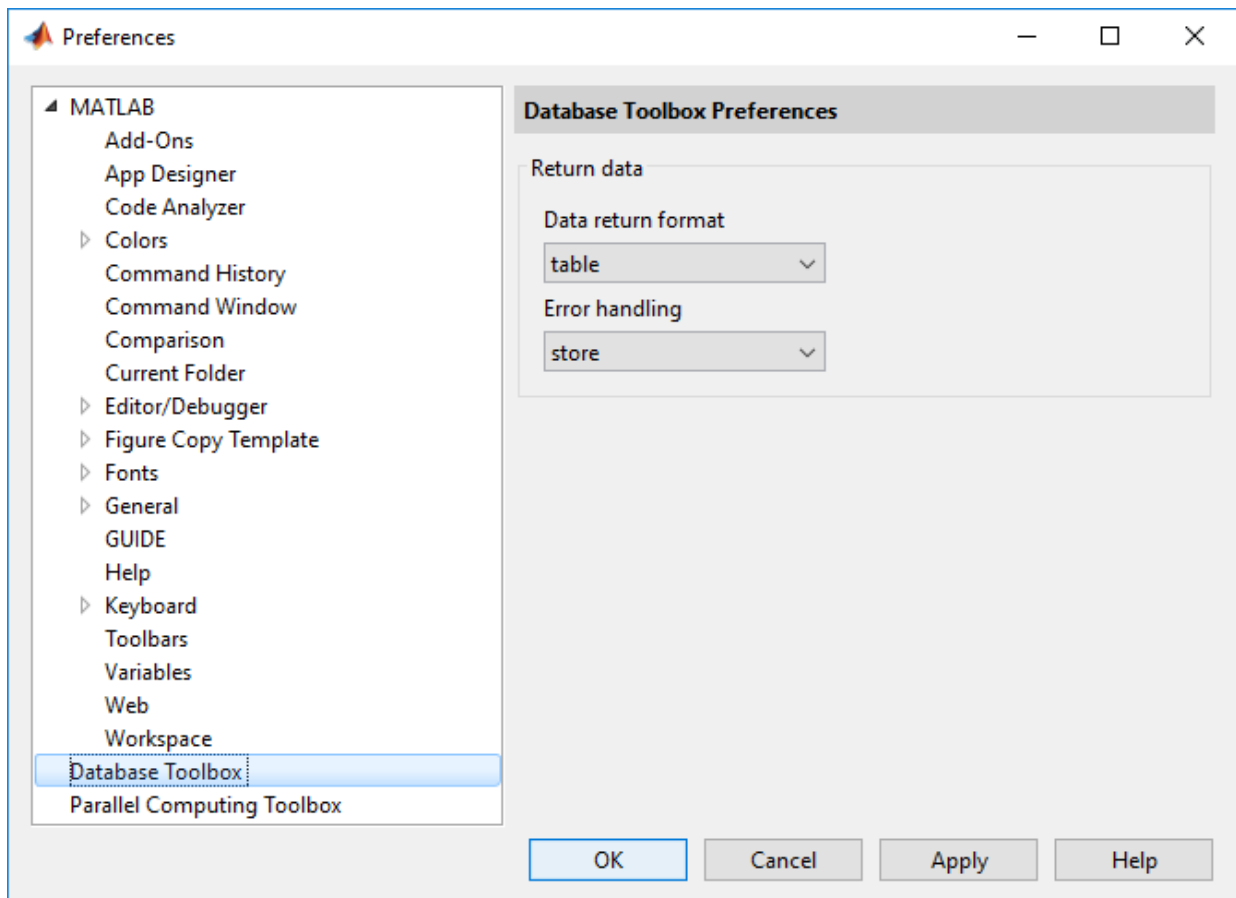
More About

- “Write Deployable MATLAB Code” (MATLAB Compiler)
- “Create Functions in Files” (MATLAB)
- “Create Standalone Application from MATLAB” (MATLAB Compiler)
- “Java Class Path” (MATLAB)

Working with Database Toolbox Preferences

Database Toolbox preferences enable you to specify the data return format of imported data and the method of error notification.

On the MATLAB Toolstrip, in the **Environment** section, click **Preferences** and select **Database Toolbox**.



Specify preferences as described in this table, and click **OK**. For details, see the `setdbprefs` function.

Preference	Acceptable Values	Description
Data return format	cellarray (default), table, numeric, or structure	<p>Specifies a data format based on the type of data you are importing, memory considerations, and your preferred method of working with imported data.</p> <ul style="list-style-type: none"> • cellarray — Imports nonnumeric data into MATLAB cell arrays. • table — Imports data into a MATLAB table. Use this value for all database data types. • numeric — Imports data into a MATLAB matrix of doubles. Nonnumeric data types are considered NULL. Use this value only when the format of the data to import is numeric, or when nonnumeric data to import is not relevant. • structure — Imports data into a MATLAB structure. Use this value for all database data types.
Error handling	store (default) or report	<p>Specifies where error messages appear after connecting to a database or importing data.</p> <ul style="list-style-type: none"> • store — Directs errors to the Message properties of the connection and cursor objects when you use the command line. • report — Displays query errors in the Command Window.

See Also

database | exec | fetch

More About

- “Configuring Driver and Data Source” on page 2-15
- “Connecting to Database” on page 2-142
- “Import Data from Databases into MATLAB” on page 5-3
- “Export Data to New Record in Database” on page 5-21

Working with Data Sources

- “Writing Data Common Errors” on page 3-2
- “Fetching Data Common Errors” on page 3-4
- “Database Connection Error Messages” on page 3-9
- “Database Explorer App Error Messages” on page 3-16
- “Connecting to Database Using Native ODBC Interface” on page 3-19

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
<i>columnname</i> 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.
<i>columnname</i> column value must be a datetime array, cell array of character vectors, or string array.		
<i>columnname</i> column value must be a logical array.		
<i>columnname</i> column value must be a cell array of character vectors or string array.		
JDBC/ODBC Error: <i>errormessage</i>	The JDBC or ODBC driver throws an error.	Consult your database driver documentation.
Unable to create <i>tablename</i> 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.

See Also

Fetching Data Common Errors

This table describes how to address common errors you might encounter while working with Database Toolbox. These errors might occur in either the Database Explorer app or the command line when using the `exec`, `fetch`, `sqlinnerjoin`, or `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
	<p>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.</p>	<p>The function cannot find common keys between the specified left and right tables to join.</p>	<p>Specify the 'Keys' name-value pair argument, or both the 'LeftKeys' and 'RightKeys' name-value pair arguments.</p>
	<p>The number of key variables on the left and right must be the same.</p>	<p>The number of specified keys for the left and right tables do not match.</p>	<p>Specify the same number of keys for the 'LeftKeys' and 'RightKeys' name-value pair arguments.</p>
<p>Microsoft SQL Server</p>	<p>The statement did not return a result set.</p>	<p>There are other SQL statements in the middle of the stored procedure. This error happens after executing <code>exec</code> but before executing <code>fetch</code>. This error happens only with the command line.</p>	<p>Add 'SET NOCOUNT ON' at the beginning of your stored procedure. For details, see <code>exec</code>.</p>
<p>Microsoft SQL Server</p>	<p>JDBC Driver 3.0 returns incorrect date values when used with JRE™ 1.7 by a Java application.</p>	<p>There is an issue with the Microsoft SQL Server JDBC Driver 3.0. This error happens after executing <code>fetch</code>. This error happens either with Database Explorer or the command line.</p>	<p>Install a hotfix from Microsoft for JDBC Driver 3.0. Alternatively, upgrade your Microsoft SQL Server JDBC driver to version 4.0.</p>

Vendor	Error Message	Probable Causes	Resolution
Microsoft SQL Server	Connection is busy with results for another command.	You are connecting to Microsoft SQL Server using a driver that preview 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 executing <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 executing <code>exec</code> but before executing <code>fetch</code> . This error happens either with Database Explorer or the command line.	<p>Write custom code. Write the code for connecting to your database via the command line. Then write the following.</p> <pre> set(conn, 'AutoCommit', 'off'); h = conn.Handle; stmt = h.createStatement(); stmt.setFetchSize(50); rs = stmt.executeQuery('SELECT * FROM largeData where productnumber <= 3000000'); </pre> <p>Modify the previous statement to include your SQL query instead.</p> <p>Then process the result set object <code>rs</code> in batches.</p>

See Also

exec | fetch

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Data Import Using Database Explorer App or Command Line” on page 2-145
- “Call Stored Procedure That Returns Data” on page 5-41

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.
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.
All JDBC-Compliant Databases	Unable to find JDBC driver file on MATLAB Java class path.	<ul style="list-style-type: none"> Path to the JDBC driver JAR file is not on the static or dynamic class path. Incorrect driver name provided while using the 'driver' and 'url' syntax. 	<p>Verify that you added the path to your JDBC driver to the static or dynamic path. For an example, see “MySQL JDBC for Windows” on page 2-61.</p> <p>Ensure that you provide the correct JDBC driver name for the driver and url input arguments. For an example, see “SQLite JDBC for Windows” on page 2-77.</p>

Vendor	Error Message	Probable Causes	Resolution
All ODBC-Compliant Databases	[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 “Configuring Driver and Data Source” on page 2-15.</p> <p>To address differences in bitness for Microsoft Access, see “Microsoft Access ODBC for Windows” on page 2-18.</p>
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	<p>Error occurs in the Connection Failure dialog box after clicking Connect in the Connect to a Data Source dialog box.</p> <p>The file location of the Microsoft Access database is incorrect.</p>	<p>Verify the location of the database file. If the database file is on a network drive, map to the network drive.</p> <p>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.</p>

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 2-32
Microsoft SQL Server	This driver is not configured for integrated authentication.	The Microsoft SQL Server Windows authentication library is not added to <code>javainlibrarypath.txt</code> .	Add the Microsoft SQL Server Windows authentication library to <code>javainlibrarypath.txt</code> . For details about configuring a Microsoft SQL Server Authenticated Database Connection, see "Microsoft SQL Server JDBC for Windows" on page 2-32.
Microsoft SQL Server	Invalid string or buffer length.	64-bit ODBC driver error.	Use a JDBC driver or the native ODBC interface instead.

Vendor	Error Message	Probable Causes	Resolution
Microsoft SQL Server	JDBC Driver Error: com.microsoft.sqlserver.jdbc.SQLServerDriver.Driver Not Found/Loaded.	The full path to the JAR file was not added to the <code>javaclasspath.txt</code> file, or it was added using the <code>javaaddpath</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	com.microsoft.sqlserver.jdbc.AuthenticationJNI <clinit> WARNING: Failed to load the sqljdbc_auth.dll	The path to the folder containing the file <code>sqljdbc_auth.dll</code> was not added to the <code>javalibrarypath.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>javalibrarypath.txt</code> file. For details about configuring a Microsoft SQL Server Authenticated Database Connection, see “Microsoft SQL Server JDBC for Windows” on page 2-32.
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.

Vendor	Error Message	Probable Causes	Resolution
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.
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 2-48.

Vendor	Error Message	Probable Causes	Resolution
Oracle	Invalid Oracle URL specified: <code>OracleDataSource.makeURL</code>	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

- “Configuring Driver and Data Source” on page 2-15
- “Connecting to Database Using Native ODBC Interface” on page 3-19

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 3-9.

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 “Configuring Driver and Data Source” on page 2-15.
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.	Add the JDBC driver to the MATLAB Java class path. For details, see “MySQL JDBC for Windows” on page 2-61.

Error Category	Error Message	Probable Causes	Resolution
SQL statements	Database Explorer supports one SELECT SQL statement only.	<p>You entered an incompatible SQL query.</p> <p>The Database Explorer app accepts a single SELECT statement only. Other SQL statements are not supported.</p>	Create one valid SQL SELECT statement in the SQL Query pane.
Data preview	Received the following message from the database:	<p>If an error occurs on the database server when it executes the SQL query, an error message appears in the Data Preview pane.</p> <p>This text precedes the exact error message from the database server.</p>	<p>Refer to the error message to determine the issue.</p> <p>Consult your database administrator for further details.</p>
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 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 4-6
- “Data Preview Using Database Explorer App” on page 4-11

Connecting to Database Using Native ODBC Interface

In this section...

“About Native ODBC Interface” on page 3-19

“Native ODBC Interface Workflow” on page 3-19

“Database Connection Type Comparison” on page 3-21

About Native ODBC Interface

When you create a database connection using an ODBC driver, Database Toolbox uses the native ODBC interface to establish the database connection. The native ODBC interface is a C++ library that allows direct communication with the ODBC driver. The native ODBC interface supports importing and exporting large amounts of data.

Native ODBC Interface Workflow

This example shows how to connect to a database using the native ODBC interface, import and export data, and close the connection.

Connect to Database Using Native ODBC Interface

Connect to the database with the ODBC data source name `dbdemo` and leave the user name and password blank.

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

`conn` is a connection object.

Import Data Using Native ODBC Interface

Select data in the column `productDescription` from `productTable` using the database connection. Assign the returned cursor object to the variable `curs`.

```
sqlquery = 'SELECT productDescription FROM productTable';  
curs = exec(conn, sqlquery);
```

With the native ODBC interface, `exec` returns `curs` as an `ODBCCursor` Object instead of a `Database Cursor` Object.

Use `fetch` to import all data into the cursor object, and store the data in a cell array contained in the cursor object property `Data`.

```
curs = fetch(curs);
```

View the contents of the `Data` property in the cursor object.

```
curs.Data
```

```
ans =
```

```
    'Victorian Doll'  
    'Train Set'  
    'Engine Kit'  
    'Painting Set'  
    'Space Cruiser'  
    'Building Blocks'  
    'Tin Soldier'  
    'Sail Boat'  
    'Slinky'  
    'Teddy Bear'
```

Export Data Using Native ODBC Interface

Define the columns of data to insert in the cell array `colnames`.

```
colnames = {'productNumber', 'stockNumber', 'supplierNumber', ...  
           'unitCost', 'productDescription'}
```

```
colnames =
```

```
Columns 1 through 3
```

```
    'productNumber'    'stockNumber'    'supplierNumber'
```

```
Columns 4 through 5
```

```
    'unitCost'    'productDescription'
```

Define the data for the row to insert in the cell array `coldata`.

```
coldata = {11,800999,1006,9.00, 'Toy Car'}
```

```
coldata =
```

```
    [11]    [800999]    [1006]    [9]    'Toy Car'
```

Insert the data in `coldata` into the table `productTable` with the defined column names.

```
datainsert(conn, 'productTable', colnames, coldata)
```

Caution: The Microsoft Access ODBC driver demonstrates unexpected behavior during large inserts. When you insert a large amount of data with Microsoft Access, insert the data in batches. For example, if you want to insert 100,000 rows of data, insert 10,000 rows at a time.

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)
close(conn)
```

Database Connection Type Comparison

You can connect to a database using an ODBC or JDBC driver with the `database` function. This table highlights the differences between using these connection types to access and manipulate data in a database.

Driver	Actions	Underlying Technology	Memory Performance
ODBC driver	Supports all actions except the <code>runstoredprocedure</code> function	C++	Restricted by MATLAB memory
JDBC driver	Supports all actions	Java	Restricted by both JVM™ heap memory and MATLAB memory

For details about choosing which connection type is best for your situation, see “Choosing Between ODBC and JDBC Drivers” on page 2-13.

See Also

More About

- “Connection Options” on page 2-9
- “Data Import Using Database Explorer App or Command Line” on page 2-145
- “Data Type Support” on page 1-3

Using Database Explorer

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Modify and Delete Data Sources” on page 4-14
- “Generate SQL Query and MATLAB Script” on page 4-17

Create SQL Queries Using Database Explorer App


Using the **Database Explorer** app, you can open one or multiple database connections simultaneously by clicking **New Query** in the toolbar. The Database Explorer app creates a data source tab for each connection.

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 in the Database Explorer app. For an example, see “MySQL ODBC for Windows” on page 2-55.
- Click a table in the **Data Browser** pane. The **SQL Query** pane in the data source tab updates with an SQL query that selects all columns and rows of the table. The **Data Preview** pane updates with a preview of the first 10 rows of data in the table.
- 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 **Data Browser** pane appears in the left **Table** list. From the left **Column** list, select the name of the shared column. Then, select the name of the table to join in the right **Table** list and select the name of the shared column in the right **Column** list. Click **Add Join**. The Database Explorer 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 4-6.
- 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 update with the chosen 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 **Edit** section, click **Clear Query** to clear the current SQL query and create a new one.
- To close the database connection, close all tabs whose names contain the data source name.

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 2-55.
- In the **Edit** 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.
- To close the database connection, close all tabs whose names contain the data source name.

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:

- 1 Click **New Query** in the toolstrip. The Connect to a Data Source dialog box opens.
- 2 Select the data source name from the **Data Source** list. Enter the user name and password for your database, and click **Connect**.
- 3 Select the catalog and schema in the Catalog and Schema dialog box, 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.

Repeat these steps to connect to the same database again and create a different SQL query. The Database Explorer app appends a numbered suffix to the data source tab name. The number increases by one for each subsequent database connection to the same data source.

Note The Microsoft Access database does not support multiple database connections.

Or, you can connect to different catalogs or schemas. Repeat these steps and select a different catalog or schema in the Catalog and Schema dialog box.

The database server settings control the maximum number of database connections you can create at a time. To increase the maximum number of database connections, contact your database administrator.

You can connect to different databases by repeating these steps and selecting a different data source name from the **Data Source** 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

- “Configuring Driver and Data Source” on page 2-15
- “Connecting to Database” on page 2-142
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11
- “Generate SQL Query and MATLAB Script” on page 4-17
- “Database Explorer App Error Messages” on page 3-16

External Websites

- SQL Tutorial

Join Tables Using Database Explorer App



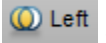

You can select and import data from multiple tables using the **Database Explorer** app. First, you must join tables, and then select the data to import. You can join tables using different join types that depend on the database.

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:

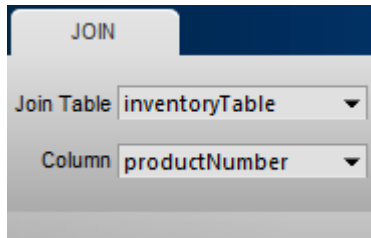
-  — An inner join retrieves records that have matching values in the selected column of both tables.
-  — A full join retrieves records that have matching values in the selected column of both tables, and unmatched records from both the left and right tables.
-  — A left join retrieves records that have matching values in the selected column of both tables, and unmatched records from the left table only.
-  — 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

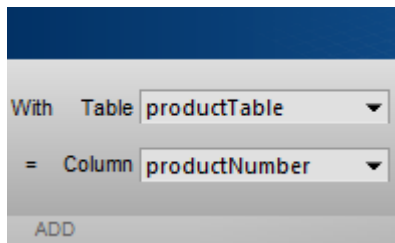
To join tables, you must know the names of each table and the names of the shared columns in the tables (that is, the primary keys). Use these steps as a general workflow for joining tables.

- 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 toolbar. 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.

Database Explorer - dbdemo

JOIN

Join Table: inventoryTable With Table: productTable

Column: productNumber = Column: productNumber

INNER JOIN inventoryTable.productNumber=

Inner Full Left Right

Remove Join Close Join

ADD EDIT CLOSE

Data Browser dbdemo

SQL 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
1	9	2339		13 2011-02-09 1...
2	8	8350		5 2011-06-18 1...
3	7	6034		16 2014-08-06 0...
4	2	1200		9 2014-07-08 2...
5	4	2580		21 2013-06-08 1...
6	1	1700	14.5000	2014-09-23 0...
7	5	9000		3 2012-09-14 1...
8	6	4540		8 2013-12-25 1...
9	3	356		17 2014-05-14 0...

Join Diagram

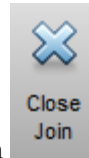
Legend: Table (green square), Join (blue circle)

Diagram: inventoryTable (Table) --- INNER JOIN --- productTable (Table)

- 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.

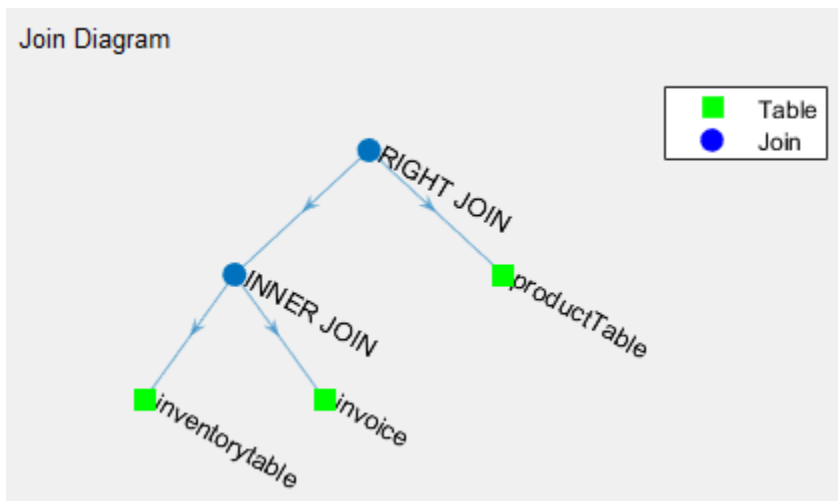
7



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 4-11
- “Generate SQL Query and MATLAB Script” on page 4-17
- “Database Explorer App Error Messages” on page 3-16

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.

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 toolstrip 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 toolstrip 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 4-6
- “Generate SQL Query and MATLAB Script” on page 4-17
- “Database Explorer App Error Messages” on page 3-16

Modify and Delete Data Sources

You can modify and delete data sources using the **Database Explorer** app. For ODBC drivers, you create data sources using the ODBC Data Source Administrator dialog box. For JDBC drivers, you create data sources using the JDBC Data Source Configuration dialog box. You can modify and delete data sources using the same dialog boxes.

Modify Data Sources

Use the Database Explorer app to modify data sources by following these steps.

ODBC Data Sources

- 1 After opening the Database Explorer app, select **Configure Data Source > Configure ODBC data source**. The ODBC Data Source Administrator dialog box opens. For details about locating this dialog box on your computer, see *Driver Installation*.

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, and close this dialog box.

JDBC Data Sources

- 1 After opening the Database Explorer app, 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 Database Explorer dialog box opens and displays a message indicating a successful connection. Otherwise, it displays an error message.

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

Delete Data Sources

Use the Database Explorer app to delete data sources by following these steps.

ODBC Data Sources

- 1 After opening the Database Explorer app, 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** and close this dialog box.

JDBC Data Sources

- 1 After opening the Database Explorer app, 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**.

The JDBC Data Source Configuration dialog box displays a message indicating a successful deletion.

- 5 Close this dialog box.

See Also

Apps

Database Explorer

More About

- “Choosing Between ODBC and JDBC Drivers” on page 2-13
- “Configuring Driver and Data Source” on page 2-15

- “Connecting to Database” on page 2-142

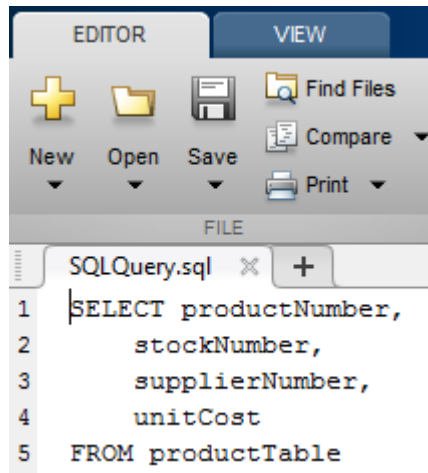
Generate SQL Query and MATLAB Script

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 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 `runsqlscript` 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 Select **Import Data > Generate MATLAB Script** to display a MATLAB script in the MATLAB Editor.

The screenshot shows the MATLAB Editor window with a script titled 'Untitled2*'. The script is a MATLAB script that automates the process of importing data from a database. The script is divided into several sections by horizontal lines, each starting with a comment line. The sections are: 1. Introduction and metadata (lines 1-9), 2. Setting preferences (lines 10-13), 3. Making a connection to the database (lines 14-16), 4. Executing a query and fetching results (lines 17-24), 5. Closing the connection to the database (lines 26-27), and 6. Restoring preferences and clearing variables (lines 29-33).

```

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 9.2 (R2017b Prerelease) and Database
8  % Toolbox Version 7.1 on 06-Apr-2017 10:30:35
9
10 %% Set preferences
11 prefs = setdbprefs('DataReturnFormat');
12 setdbprefs('DataReturnFormat','table')
13
14 %% Make connection to database
15 conn = database('MS SQL Server Auth','','');
16
17 %% Execute query and fetch results
18 curs = exec(conn,['SELECT productNumber, ' ...
19     '   Quantity, ' ...
20     '   Price ' ...
21     'FROM toy_store.dbo.inventorytable']);
22 curs = fetch(curs);
23 data = curs.Data;
24 close(curs)
25
26 %% Close connection to database
27 close(conn)
28
29 %% Restore preferences
30 setdbprefs('DataReturnFormat',prefs)
31
32 %% Clear variables
33 clear prefs conn curs

```

- 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” (MATLAB).

See Also

Functions

`runsqlscript`

Apps

Database Explorer

More About

- “Create SQL Queries Using Database Explorer App” on page 4-2
- “Join Tables Using Database Explorer App” on page 4-6
- “Data Preview Using Database Explorer App” on page 4-11

Using Database Toolbox Functions

- “Import Data from Databases into MATLAB” on page 5-3
- “Create Queries with Characters and Variables” on page 5-7
- “Roll Back and Commit Data in Database” on page 5-12
- “Change Database Connection Catalog” on page 5-13
- “Create Table and Add Column” on page 5-14
- “Delete Data from Databases” on page 5-15
- “Roll Back Data After Updating Record” on page 5-18
- “Export Data to New Record in Database” on page 5-21
- “Replace Existing Data in Database” on page 5-25
- “Export Multiple Records from MATLAB Workspace” on page 5-27
- “Export Data Using Bulk Insert” on page 5-32
- “Display Database Metadata” on page 5-38
- “Call Stored Procedure That Returns Data” on page 5-41
- “Run Custom Database Function” on page 5-44
- “Data Import Approaches and Memory Management” on page 5-46
- “Display Information About Imported Data” on page 5-50
- “Using Scrollable Cursors” on page 5-53
- “Import Data Using Scrollable Cursor with Relative Position Offset” on page 5-61
- “Import Large Data Using DatabaseDatastore Object” on page 5-64
- “Import Data Using MATLAB® Interface to SQLite” on page 5-68
- “Retrieve Image Data Types” on page 5-73
- “Import Boolean Data from Database” on page 5-78
- “Append Data to Existing Database Table Using Insert Functionality” on page 5-80
- “Insert Data into New Database Table Using Insert Functionality” on page 5-82
- “Join Tables Using Command Line” on page 5-84
- “Import Data from Database Table Using sqlread Function” on page 5-85

- “Insert Data into Database Table” on page 5-88
- “Retrieve Database Metadata” on page 5-91

Import Data from Databases into MATLAB

This example shows how to import data from a Microsoft Access database called `dbdemo` into the MATLAB workspace.

Connect to Database

Connect to the Microsoft Access database with the data source name `dbdemo` using an ODBC driver.

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

If you are connecting to a database using a JDBC connection, then specify a different syntax for the database function.

Import Data Using Simple SQL Query

Select the product number `productNumber` and description `productDescription` from the product table `productTable`. Create an SQL query to select this data. Then, use the `exec` function to execute the SQL query using the database connection.

```
sqlquery = 'SELECT productNumber,productDescription FROM productTable';  
curs = exec(conn,sqlquery);
```

The data contains text. Set the data return format to support text. Specify the format `cellarray` using `setdbprefs`.

```
setdbprefs('DataReturnFormat','cellarray')
```

Display the data. Import the data from the executed SQL query using `fetch`. The `Data` property of the cursor object `curs` contains the data.

```
curs = fetch(curs);  
curs.Data
```

```
ans =
```

```
[ 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'
```

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

Import Data Using Multiple Joins in SQL Query

Connect to the Microsoft Access database with the data source name `dbdemo` using an ODBC driver.

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

Create an SQL script file named `salesvolume.sql` with this 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 SQL script file named `salesvolume.sql` using the `runsqlscript` function.

```
results = runsqlscript(conn, 'salesvolume.sql');
```

`results` is a cursor object array with the returned data from running the SQL query in the SQL script file.

Display the data in the cursor object containing the returned data.

```
results(1).Data
```

```
ans =
```

```
Columns 1 through 8
```

```
[5000.00] [3500.00] [2800.00] [2300.00] [1700.00] [1400.00] [1000.00] [900.00]
[2400.00] [1721.00] [1414.00] [1191.00] [ 983.00] [ 825.00] [ 731.00] [653.00]
[1200.00] [ 900.00] [ 800.00] [ 500.00] [ 399.00] [ 345.00] [ 300.00] [175.00]
...
```

```
Columns 9 through 13
```

```
[1600.00] [3300.00] [12000.00] [20000.00] 'United States'
[ 723.00] [ 790.00] [ 1400.00] [ 5000.00] 'United States'
[ 760.00] [1500.00] [ 5500.00] [17000.00] 'United States'
...
```

Display the column names for the returned data.

```
columnnames(results(1))
```

```
ans =
```

```
'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', ...
'September', 'October', 'November', 'December', 'Country'
```

Close Database Connection

Close the cursor object array and database connection.

```
close(results)
close(conn)
```

See Also

`close` | `database` | `exec` | `fetch` | `runsqlscript` | `setdbprefs`

Related Examples

- “Create Queries with Characters and Variables” on page 5-7

More About

- “Configuring Driver and Data Source” on page 2-15
- “Connecting to Database Using Native ODBC Interface” on page 3-19

External Websites

- [SQL Tutorial](#)

Create Queries with Characters and Variables

The following examples show how to create queries using a date, text, a MATLAB variable, and special characters. Construct these queries using the command line.

Create Query Using Date

This example shows how to format a date in an SQL query.

When you want to write an SQL statement that selects data from your database using a date, format the date according to your database specifications. Consult your database documentation for the right formatting. This example shows date formatting for an Oracle database.

Create the database connection `conn` to an Oracle database using an ODBC driver. For example, the following code assumes that you are connecting to a data source named Oracle with user name `username` and password `pwd`.

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

Create an SQL statement `sqlquery` that contains the full query. Execute the query using `conn`. The following code uses the table `test_types` and the column `test_dt`. The WHERE clause contains Oracle SQL code for filtering the records based on the date. The `test_dt` column data type is an Oracle date type. Filter records for the dates after June 9, 2013 using the `test_dt` column. To convert your date to an Oracle date type, enter this date in the Oracle function `to_date`. For a date '2013-06-09', specify the format as 'YYYY-MM-DD'. 'YYYY-MM-DD' is one way to format a date in Oracle. Consult your Oracle documentation for alternatives.

```
sqlquery = ['SELECT * FROM test_types ' ...
           'where test_dt > to_date(''2013-06-09'', 'YYYY-MM-DD')'];
curs = exec(conn,sqlquery);
```

Import the data using the cursor object `curs`. The `Data` property of `curs` contains the imported data. Display the data.

```
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
'2013-06-10 15:11:00'      '2013-06-10 15:11:22.500000'
```

```
'2013-06-10 15:13:00'      '2013-06-10 15:13:21.870003'  
'2013-06-10 15:16:00'      '2013-06-10 15:16:45.099998'  
...
```

The query returns the records where the date in the column `test_dt` is after June 9, 2013.

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

Create Query Using Text

This example shows how to include text in your SQL query using a Microsoft Access database.

Create the database connection `conn` to a Microsoft Access database using an ODBC driver. For example, the following code assumes that you are connecting to a data source named `dbdemo` with a blank user name and password.

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

Select all records from the table `productTable` where the product description is 'Slinky'. Create an SQL query `sqlquery` that embeds the product description into the SQL query by using an extra pair of single quotes.

```
sqlquery = ['SELECT * FROM productTable ' ...  
           'where productDescription = 'Slinky'''];
```

Or, you can write the SQL query as a concatenation of two character vectors using brackets.

```
sqlquery = ['SELECT * FROM productTable ' ...  
           'where productDescription = ' ''Slinky'''];
```

Execute the SQL query `sqlquery` using `conn`. The `cursor` object `curs` contains the executed query. Import the data from the executed query using the `fetch` function. The `Data` property of `curs` contains the imported data. Display the data.

```
curs = exec(conn,sqlquery);  
curs = fetch(curs);  
curs.Data
```

```
ans =
      [3.00]      [400999.00]      [1009.00]      [17.00]      'Slinky'
```

Data contains the product record where the product description is 'Slinky'.

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)
close(conn)
```

Create Query Using MATLAB Variable

This example shows how to include a MATLAB variable in your SQL query. This example uses a Microsoft SQL Server database.

Create the database connection `conn` to a Microsoft SQL Server database using a JDBC driver without operating system authentication. For example, this code assumes that you are connecting to a database named `dbname` with the user name `username`, password `pwd`, database server name `sname`, and port number 123456.

```
conn = database('dbname','username','pwd', ...
    'Vendor','Microsoft SQL Server','Server','sname', ...
    'AuthType','Server','PortNumber',123456);
```

Suppose that you want to select all invoice data for the first product. Create a MATLAB variable `productID` and set it to the first product number.

```
productID = 1;
```

Select all records from the table `invoice` where the product number is equal to the first product. Create an SQL query `sqlquery` that concatenates the SQL query with the MATLAB variable `productID` by using brackets. `productID` is a numeric variable but the SQL query is a character vector. You need convert the number to a character vector by using the `num2str` function.

```
sqlquery = ['SELECT * FROM invoice ' ...
    'where ProductNumber = ' num2str(productID)];
```

Execute the SQL query `sqlquery` using `conn`. The `cursor` object `curs` contains the executed query. Import the data from the executed query using the `fetch` function. The `Data` property of `curs` contains the imported data. Display the data.

```
curs = exec(conn,sqlquery);
curs = fetch(curs);
curs.Data

ans =

    [2101.00]    '2010-08-01 00:00...'    [1.00]    [0]    [1948410x1 int8]
```

Data contains the invoice data record for the first product.

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)
close(conn)
```

Create Query Using Special Characters

This example shows how to write an SQL query for table names or columns names with special characters.

These characters require using escape characters that are specific to your database. Consult your database documentation for the right escape characters. This example uses a Microsoft SQL Server database.

Create the database connection `conn` to a Microsoft SQL Server database using a JDBC driver without operating system authentication. For example, this code assumes that you are connecting to a database named `dbname` with the user name `username`, password `pwd`, database server name `sname`, and port number 123456.

```
conn = database('dbname','username','pwd', ...
    'Vendor','Microsoft SQL Server','Server','sname', ...
    'AuthType','Server','PortNumber',123456);
```

Suppose that you want to select all data in a column with a column name that contains spaces. This column resides in a table with a table name that contains spaces. A space is a special character. Enclose spaces with escape characters so that the SQL query executes. Brackets are the escape characters for a Microsoft SQL Server database. Create an SQL query `sqlquery` that contains the column name and table name enclosed by brackets.

```
sqlquery = 'SELECT [column with spaces] FROM [table with spaces]';
```

Execute the SQL query `sqlquery` using `conn`. The `cursor` object `curs` contains the executed query. Import the data from the executed query using the `fetch` function. The `Data` property of `curs` contains the imported data. Display the data.

```
curs = exec(conn,sqlquery);  
curs = fetch(curs);  
curs.Data  
  
ans =  
  
    'some text'  
    'some text'
```

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

See Also

`close` | `database` | `exec` | `fetch` | `num2str`

Related Examples

- “Import Data from Databases into MATLAB” on page 5-3

More About

- “Connecting to Database Using Native ODBC Interface” on page 3-19

External Websites

- SQL Tutorial

Roll Back and Commit Data in Database

This example assumes that you have established a connection to the database named `conn`. Use `exec` to roll back and commit data after running `datainsert`, `fastinsert`, `insert`, or `update` for which the `AutoCommit` flag is off.

Roll back data for the database connection `conn`.

```
sqlquery = 'rollback';  
exec(conn,sqlquery)
```

When you do not specify an output argument, MATLAB returns the results of calling `exec` into cursor object `ans`. Assign `ans` to variable `curs` so that MATLAB does not overwrite the cursor object. After you finish working with the cursor object, close it.

```
curs = ans;  
close(curs)
```

Commit the data.

```
sqlquery = 'commit';  
exec(conn,sqlquery)
```

After you finish working with the cursor object, close it.

```
curs = ans;  
close(curs)
```

See Also

[close](#) | [database](#) | [exec](#)

Related Examples

- “Export Data to New Record in Database” on page 5-21
- “Replace Existing Data in Database” on page 5-25

Change Database Connection Catalog

This example assumes that you have established a connection to the database named `conn`. You can work with data in a different catalog within the same database using `exec`.

Change the catalog for the database connection `conn` to `intlprice`. The cursor object `curs` contains the executed query.

```
sqlquery = 'Use intlprice';  
curs = exec(conn,sqlquery);
```

After you finish working with the cursor object, close it.

```
close(curs)
```

See Also

[close](#) | [database](#) | [exec](#)

Related Examples

- “Display Database Metadata” on page 5-38

External Websites

- [SQL Tutorial](#)

Create Table and Add Column

This example assumes that you have established a connection to the database named `conn`. You can manipulate the database structure using `exec`.

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. The cursor object contains the executed query.

```
curs = exec(conn,sqlquery);
```

After you finish working with the cursor object, close it.

```
close(curs)
```

Use the SQL `ALTER` statement to add the column `City` to the table `Person`.

```
sqlquery = 'ALTER TABLE Person ADD City varchar(30)';
```

```
curs = exec(conn,sqlquery);
```

After you finish working with the cursor object, close it.

```
close(curs)
```

See Also

[close](#) | [database](#) | [exec](#)

Related Examples

- “Display Database Metadata” on page 5-38

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 `exec` 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`. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function. The `Data` property of `curs` contains the imported data. Display the data.

```
sqlquery = 'SELECT * FROM inventoryTable';
```

```
curs = exec(conn,sqlquery);
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
...
[11] [ 567] [ 0] '2012-09-11 00:30...'
[12] [1278] [ 0] '2010-10-29 18:17...'
[13] [1700] [14.5000] '2009-05-24 10:58...'
```

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';
```

```
curs = exec(conn,sqlquery);
```

Display the data in the table `inventoryTable` after the deletion.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data

ans =

    ...
    [10]    [ 723]    [    24]    '2012-03-14 13:13...'
    [11]    [ 567]    [     0]    '2012-09-11 00:30...'
    [12]   [1278]    [     0]    '2010-10-29 18:17...'
```

The record with product number 13 is missing.

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)];
```

```
curs = exec(conn, sqlquery);
```

Display the data in the table `inventoryTable` after the deletion.

```
sqlquery = 'SELECT * FROM inventoryTable';
curs = exec(conn, sqlquery);
curs = fetch(curs);
curs.Data

ans =

    ...
    [ 9]    [2339]    [    13]    '2011-02-09 12:50...'
   [10]    [ 723]    [    24]    '2012-03-14 13:13...'
   [11]    [ 567]    [     0]    '2012-09-11 00:30...'
```

The record with product number 12 is missing.

Close Cursor and Database Connection

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)
close(conn)
```

See Also

`exec` | `fetch` | `num2str`

Related Examples

- “Import Data from Databases into MATLAB” on page 5-3

More About

- “Connecting to Database Using Native ODBC Interface” on page 3-19

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` flag to `off`. Any updates you make after turning off this flag do not commit to the database automatically.

```
set(conn,'AutoCommit','off')
```

Display the data in the `inventoryTable` table before making updates. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
curs = exec(conn,'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
    [ 1]    [ 1700]    [14.5000]    '2014-10-20 00:00...'
    [ 2]    [ 1200]    [ 9.3000]    '2014-10-20 00:00...'
    [ 3]    [  356]    [17.2000]    '2014-10-20 00:00...'
    ...
```

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.

```
curs = exec(conn,'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data

ans =

     [ 1]     [ 1700]     [      30]     '2014-10-20 00:00...'
     [ 2]     [ 1200]     [  9.3000]     '2014-10-20 00:00...'
     [ 3]     [   356]     [17.2000]     '2014-10-20 00:00...'
     ...
```

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 with the `exec` function.

Display the data in the `inventoryTable` table after rolling back the update.

```
curs = exec(conn,'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data

ans =

     [ 1]     [ 1700]     [14.5000]     '2014-10-20 00:00...'
     [ 2]     [ 1200]     [  9.3000]     '2014-10-20 00:00...'
     [ 3]     [   356]     [17.2000]     '2014-10-20 00:00...'
     ...
```

The first product has the old price of 14.50.

After you finish working with the cursor object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

See Also

`close` | `database` | `exec` | `fetch` | `rollback` | `set`

Related Examples

- “Export Data to New Record in Database” on page 5-21
- “Replace Existing Data in Database” on page 5-25

External Websites

- [SQL Tutorial](#)

Export Data to New Record in Database

This example does the following:

- 1 Retrieves sales data from a `salesVolume` table.
- 2 Calculates the sum of sales for 1 month.
- 3 Stores this data in a cell array.
- 4 Exports this data to a `yearlySales` table.

This example assumes that you are connecting to a Microsoft Access database that contains tables named `salesVolume` and `yearlySales`. The table `salesVolume` contains the column names for each month. The table `yearlySales` contains the column names `Month` and `salesTotal`.

To access the code for this example, see `matlab\toolbox\database\dbdemos\dbinsertdemo.m`.

Note The `datainsert` function will be removed in a future release. Use the `sqlwrite` function instead.

- 1 Create a database connection `conn` to the Microsoft Access database. For example, the following code assumes that you are connecting to a data source named `dbdemo` with blank user name and password.

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

- 2 Set the format for retrieved data to `numeric` by using `setdbprefs`.

```
setdbprefs('DataReturnFormat', 'numeric')
```

- 3 Execute the SQL query `sqlquery` using `conn` to import data for the `March` column from the `salesVolume` table. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
sqlquery = 'SELECT March FROM salesVolume';
```

```
curs = exec(conn, sqlquery);  
curs = fetch(curs);
```

- 4 The `Data` property of `curs` contains the imported data. Assign the data to the MATLAB workspace variable `AA`. Display the data.

```
AA = curs.Data
```

```
AA =
```

```
981  
1414  
890  
1800  
2600  
2800  
800  
1500  
1000  
821
```

- 5 Calculate the sum of the March sales. Assign the result to the MATLAB workspace variable `sumA`. Display the sum.

```
sumA = sum(AA(:))
```

```
sumA =
```

```
14606
```

- 6 To export the data to the database, assign the month and sum of sales to a cell array. Put the month in the first cell of cell array `exdata`. Put the sum in the second cell of `exdata`.

```
exdata(1,1) = {'March'};  
exdata(1,2) = {sumA}
```

```
exdata =
```

```
'March' [14606]
```

- 7 Define the names of the columns. Assign the cell array containing the column names to the MATLAB workspace variable `colnames`.

```
colnames = {'Month', 'salesTotal'};
```

- 8 Determine the status of the `AutoCommit` database flag using `get`. This status determines if the exported data automatically commits to the database. If the flag is off, you can undo an insert. If the flag is on, data automatically commits to the database.

```
get(conn, 'AutoCommit')
```

```
ans =
```

```
on
```

The `AutoCommit` flag is set to on. The exported data automatically commits to the database.

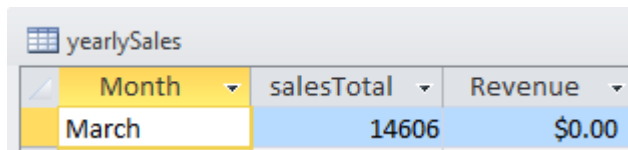
- 9 Export the data into the `yearlySales` table using these arguments:

- Database connection `conn`
- Table name `yearlySales`
- Column names `colnames`
- Export data `exdata`

```
datainsert(conn, 'yearlySales', colnames, exdata)
```

`datainsert` appends the data as a new record at the end of the `yearlySales` table.

- 10 In Microsoft Access, view the `yearlySales` table to verify the results.



Month	salesTotal	Revenue
March	14606	\$0.00

- 11 After you finish working with the cursor object, close it.

```
close(curs)
```

- 12 Close the database connection.

```
close(conn)
```

See Also

`datainsert` | `get` | `setdbprefs`

Related Examples

- “Export Multiple Records from MATLAB Workspace” on page 5-27
- “Export Data Using Bulk Insert” on page 5-32
- “Replace Existing Data in Database” on page 5-25

More About

- “Inserting Data Using Command Line” on page 2-148

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 “Export Data to New Record in Database” on page 5-21.

To access the code for this example, see `matlab\toolbox\database\dbdemos\dbupdatedemo.m`.

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'''
```

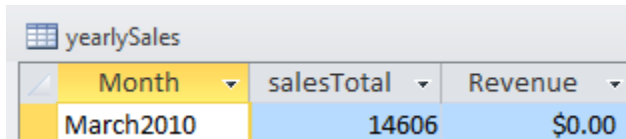
```
whereclause =
```

```
    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

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

See Also

[close](#) | [database](#) | [update](#)

Related Examples

- “Export Data to New Record in Database” on page 5-21
- “Export Multiple Records from MATLAB Workspace” on page 5-27

External Websites

- [SQL Tutorial](#)

Export Multiple Records from MATLAB Workspace

This example does the following:

- 1 Imports monthly sales figures for all products from the `dbdemo` data source into the MATLAB workspace.
- 2 Computes total sales for each month.
- 3 Exports the totals to a new table.

This example assumes that you are connecting to a Microsoft Access database that contains tables named `salesVolume` and `yearlySales`. The table `salesVolume` contains the column names for each month. The table `yearlySales` contains the column named `salesTotal`.

To access the code for this example, see `matlab\toolbox\database\dbdemos\dbinsert2demo.m`.

Note The `datainsert` function will be removed in a future release. Use the `sqlwrite` function instead.

- 1 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', '', '');
```

- 2 Ensure that the database is writable using `conn`.

```
a = isreadonly(conn)
```

```
a =  
0
```

When the `isreadonly` function returns `0`, the database is writable.

- 3 Specify preferences for the retrieved data. Set the data return format to `numeric`. Specify that `NULL` values read from the database are converted to `0` in the MATLAB workspace.

```
setdbprefs...  
({'NullNumberRead'; 'DataReturnFormat'}, {'0'; 'numeric'})
```

When you specify `DataReturnFormat` as `numeric`, the value for `NullNumberRead` must be `numeric`.

- 4 Execute the SQL query `sqlquery` using `conn` to import all data from the `salesVolume` table. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
sqlquery = 'SELECT * FROM salesVolume';
```

```
curs = exec(conn,sqlquery);
```

```
curs = fetch(curs);
```

- 5 Display the names of the columns in the fetched data set.

```
columnnames(curs)
```

```
ans =
```

```
    'StockNumber', 'January', 'February', 'March', 'April',  
    'May', 'June', 'July', 'August', 'September', 'October',  
    'November', 'December'
```

- 6 Display the data for January. January data is in the second column of the fetched data set.

```
curs.Data(:,2)
```

```
ans =
```

```
    1400  
    2400  
    1800  
    3000  
    4300  
    5000  
    1200  
    3000  
    3000  
     0
```

- 7 Assign the dimensions of the matrix containing the fetched data set to `m` and `n`.

```
[m,n] = size(curs.Data)
```

```
m =
```

```
    10
```

```
n =
```

```
    13
```


- 8** Calculate monthly totals using `m` and `n`. The variable `tmp` is the sales volume for all products in a given month `c`. The variable `monthly` is the total sales volume of all products for that month. For example, if `c` is 2, row 1 of `monthly` is the total of all rows in column 2 of `curs.Data`, where column 2 is the sales volume for January.

```
for c = 2:n
    tmp = curs.Data(:,c);
    monthly(c-1,1) = sum(tmp(:));
end
```

- 9** Display the monthly totals.

```
monthly
```

```
ans =
```

```
25100
15621
14606
11944
9965
8643
6525
5899
8632
13170
48345
172000
```

- 10** Create a cell array `colnames` containing the column name for inserting the data.

```
colnames{1,1} = 'salesTotal';
```

- 11** Insert the data into the `yearlySales` table using `conn`, `colnames`, and the monthly totals `monthly`.

```
datainsert(conn,'yearlySales',colnames,monthly)
```

- 12** To verify the data import in Microsoft Access, view the `yearlySales` table from the `tutorial` database.

	Month	salesTotal	Revenue
▶		25100	\$0.00
		15621	\$0.00
		14606	\$0.00
		11944	\$0.00
		9965	\$0.00
		8643	\$0.00
		6525	\$0.00
		5899	\$0.00
		8632	\$0.00
		13170	\$0.00
		48345	\$0.00
		172000	\$0.00
*		0	\$0.00

Record: 1 of 12

- 13 After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

See Also

database | datainsert | exec | fetch | isreadonly | setdbprefs

Related Examples

- “Export Data to New Record in Database” on page 5-21

More About

- “Inserting Data Using Command Line” on page 2-148

External Websites

- [SQL Tutorial](#)

Export Data Using Bulk Insert

Bulk Insert Functionality

You can insert data into your database using any one of these functions at the command line: `datainsert`, `fastinsert`, or `insert`. However, for best performance with large volumes of data, use `datainsert` or `fastinsert`. For differences between these functions, see “Inserting Data Using Command Line” on page 2-148.

If you still experience performance issues, create a data file with 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 2-150.

Bulk Insert into Oracle

This example uses a data file on the local machine where Oracle is installed and exports data to the Oracle server using bulk insert functionality.

- 1 Connect to the Oracle database.

```
javaaddpath 'path\ojdbc5.jar';  
conn = database('databasename','user','password', ...  
    'oracle.jdbc.driver.OracleDriver', ...  
    'jdbc:oracle:thin:@machine:port:databasename');
```

- 2 Create a table named BULKTEST.

```
curs = exec(conn,['CREATE TABLE BULKTEST (salary number, ' ...  
    'player varchar2(25), signed varchar2(25), ' ...  
    'team varchar2(25)')']);  
close(curs)
```

- 3 Create a data record.

```
A = {100000.00, 'KGreen', '06/22/2011', 'Challengers'};
```

- 4 Expand A to a 10,000-record data set.

```
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 instance of 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.

```
curs = exec(conn, ...
    'CREATE OR REPLACE DIRECTORY ext AS 'C:\\Temp''');
close(curs)
```

- 7 Delete the temporary table, if it exists.

```
curs = exec(conn,'DROP TABLE testinsert');
try,close(curs),end
```

- 8 Create a temporary table and use bulk insert functionality to insert it into the table BULKTEST.

```
curs = exec(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']);
close(curs)
curs = exec(conn,'INSERT INTO BULKTEST SELECT * FROM testinsert');
close(curs)
```

- 9 Confirm the number of rows and columns in BULKTEST.

```
curs = exec(conn,'SELECT * FROM BULKTEST');
curs = fetch(curs);
columnnames(curs)
```

```
ans =
```

```
'SALARY', 'PLAYER', 'SIGNED', 'TEAM'
```

- 10 After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)
close(conn)
```

Bulk Insert into Microsoft SQL Server 2005

This example uses a data file on the local machine where Microsoft SQL Server is installed and exports data to the Microsoft SQL Server using bulk insert functionality.

- 1 Connect to the Microsoft SQL Server. For JDBC driver use, add the JAR file to the MATLAB Java class path.

```
javaaddpath 'path\sqljdbc4.jar';
conn = database('databasename','user','password', ...
    'com.microsoft.sqlserver.jdbc.SQLServerDriver', ...
    'jdbc:sqlserver://machine:port;database=databasename');
```

- 2 Create a table named BULKTEST.

```
curs = exec(conn,['CREATE TABLE BULKTEST (salary ' ...
    'decimal(10,2), player varchar(25), signed_date ' ...
    'datetime, team varchar(25))']);
close(curs)
```

- 3 Create a data record.

```
A = {100000.00, 'KGreen', '06/22/2011', 'Challengers'};
```

- 4 Expand A to a 10,000-record data set.

```
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 instance of 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.

```
curs = exec(conn,['BULK INSERT BULKTEST FROM ' ...
    'c:\temp\tmp.txt' WITH (FIELDTERMINATOR = '\t', ' ...
    'ROWTERMINATOR = '\n')]);
close(curs)
```

- 7 Confirm the number of rows and columns in BULKTEST.

```
curs = exec(conn,'SELECT * FROM BULKTEST');
curs = fetch(curs);
columnnames(curs)
```

```
ans =
```

```
'salary','player','signed_date','team'
```

- 8 After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)
close(conn)
```

Bulk Insert into MySQL

This example uses a data file on the local machine where MySQL is installed and exports data to a MySQL database using bulk insert functionality.

- 1 Connect to the MySQL server. For JDBC driver use, add the JAR file to the MATLAB Java class path.

```
javaaddpath 'path\mysql-connector-java-5.1.13-bin.jar';
conn = database('databasename', 'user', 'password', ...
    'com.mysql.jdbc.Driver', ...
    'jdbc:mysql://machine:port/databasename');
```

- 2 Create a table named BULKTEST.

```
curs = exec(conn,['CREATE TABLE BULKTEST (salary decimal, ' ...
    'player varchar(25), signed_date varchar(25), ' ...
    'team varchar(25))']);
close(curs)
```

- 3 Create a data record.

```
A = {100000.00, 'KGreen', '06/22/2011', 'Challengers'};
```

- 4 Expand A to be a 10,000-record data set.

```
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.

```
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. Note the use of the SQL statement LOCAL INFILE.

```
curs = exec(conn,['LOAD DATA LOCAL INFILE ' ...
    ' 'C:\temp\tmp.txt' INTO TABLE BULKTEST ' ...
    ' FIELDS TERMINATED BY '\t' LINES TERMINATED ' ...
    ' BY '\n'']);
close(curs)
```

- 7 Confirm the number of rows and columns in BULKTEST.

```
curs = exec(conn, 'SELECT * FROM BULKTEST');
results = fetch(curs)
columnnames(curs)
```

```
ans =
```

```
'salary','player','signed_date','team'
```

- 8 After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)
close(conn)
```

See Also

[close](#) | [database](#) | [exec](#) | [fetch](#)

Related Examples

- “Export Data to New Record in Database” on page 5-21

More About

- “Inserting Data Using Command Line” on page 2-148

External Websites

- SQL Tutorial

Display Database Metadata

This example shows how to display database information for `connection` objects using the command line. To view the database structure quickly, use the **Database Explorer** app to explore the tables and column names. Here, metadata refers to the information about the database structure and various database properties.

Note The `dmd` object will be removed in a future release. Use the properties of the `connection` object and the `sqlfind` function instead.

Create Database Connection

Create a database connection `conn` using the `dbdemo` data source.

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

Determine if the database connection `conn` is open.

```
o = isopen(conn)
```

```
o =
```

```
1
```

`o` returns as the scalar `1` that denotes the database connection is open.

Create Database Metadata Object

Create a database metadata object `dbmeta` using `conn`.

```
dbmeta = dmd(conn)
```

```
dbmeta =
```

```
  dmd with properties:
```

```
  DMDHandle: [1x1 database.internal.ODBCDatabaseMetadataHandle]
```

Display Database Properties

Display the database properties `dbprops` of the database metadata object `dbmeta`.

```
dbprops = get(dbmeta)
```

```
dbprops =
    AllProceduresAreCallable: 1
    AllTablesAreSelectable: 1
    DataDefinitionCausesTransactionCommit: 1
    ...
```

For details about the database metadata properties returned by `get`, see the methods of the `DatabaseMetaData` object on the Oracle Java website:

<http://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html>.

Retrieve Catalog Metadata

Retrieve the names and types of tables in a catalog in the database using `dbmeta` and the catalog name `tutorial`.

```
t = tables(dbmeta, 'tutorial')
t =
    'MSysAccessObjects'    'SYSTEM TABLE'
    'MSysIMEXColumns'     'SYSTEM TABLE'
    'MSysIMEXSpecs'       'SYSTEM TABLE'
    'MSysObjects'         'SYSTEM TABLE'
    'MSysQueries'          'SYSTEM TABLE'
    'MSysRelationships'   'SYSTEM TABLE'
    'inventoryTable'      'TABLE'
    'productTable'        'TABLE'
    'salesVolume'         'TABLE'
    'suppliers'           'TABLE'
    'yearlySales'         'TABLE'
    'display'             'VIEW'
```

`t` contains the list of table names in the catalog in the first column and list of table types in the second column.

Close Database Connection

`close(conn)`

See Also

`dmd` | `get` | `supports` | `tables`

Related Examples

- “Display Information About Imported Data” on page 5-50

Call Stored Procedure That Returns Data

This example shows how to call a stored procedure that returns data using the `exec` 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 fetch the results of the `SELECT` query.

Use `exec` 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 located on the machine `servername`.

```
conn = database('test_db', 'username', 'pwd', ...
    'Vendor', 'Microsoft SQL Server', ...
    'Server', 'servername', 'PortNumber', 1234);
```

Call Stored Procedure

Return the result set in table format by using `setdbprefs` to set `'DataReturnFormat'` to `'table'`.

```
setdbprefs('DataReturnFormat', 'table')
```

Call the stored procedure, `getSupplierInfo`, to return supplier information for New York city using `exec` and the database connection.

```
sqlquery = '{call getSupplierInfo(''New York'')}';  
curs = exec(conn,sqlquery)
```

```
curs =  
  
    cursor with properties:  
  
    Attributes: []  
        Data: 0  
    DatabaseObject: [1x1 database.jdbc.connection]  
    RowLimit: 0  
    SQLQuery: '{call getSupplierInfo(''New York'')}'  
    Message: []  
        Type: 'Database Cursor Object'  
    ResultSet: [1x1 com.microsoft.sqlserver.jdbc.SQLServerResultSet]  
        Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]  
    Statement: [1x1 com.microsoft.sqlserver.jdbc.SQLServerStatement]  
    Fetch: 0
```

`exec` returns a cursor object that contains the supplier information.

Retrieve Output Data from Stored Procedure

Retrieve supplier data from the cursor object using the `fetch` function.

```
curs = fetch(curs)
```

```
curs =  
  
    cursor with properties:  
  
    Attributes: []  
        Data: [3x5 table]  
    DatabaseObject: [1x1 database.jdbc.connection]  
    RowLimit: 0  
    SQLQuery: '{call getSupplierInfo(''New York'')}'  
    Message: []  
        Type: 'Database Cursor Object'  
    ResultSet: [1x1 com.microsoft.sqlserver.jdbc.SQLServerResultSet]  
        Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]  
    Statement: [1x1 com.microsoft.sqlserver.jdbc.SQLServerStatement]  
    Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

`curs` contains the supplier data obtained from calling the stored procedure, `getSupplierInfo`.

Display the supplier data in table format by accessing the contents of the `Data` property of the cursor object.

```
curs.Data
```

```
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

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

See Also

[database](#) | [exec](#) | [fetch](#) | [runstoredprocedure](#) | [setdbprefs](#)

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 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 by running `exec`. The cursor object `curs` contains the results from executing the custom function. Import the data from the custom function using the `fetch` function.

```
sqlquery = 'SELECT dbo.get_prodCount() as num_products';
```

```
curs = exec(conn, sqlquery);  
curs = fetch(curs);
```

Display the result.

```
curs.Data
```



```
ans =  
    [30.00]
```

The custom function `get_prodCount` returns the product count 30.

Close Database Connection

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

See Also

`close` | `database` | `exec` | `fetch`

External Websites

- [SQL Tutorial](#)

Data Import Approaches and 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 2-145.

The way you import data has a different impact on memory use. You can import data in one or two steps. Also, you can import large data by limiting the number of rows.

Data Import in One Step

If you are not familiar with writing SQL queries, you can import data in one step using the `sql read` function. This function needs only a database connection and the database table name to import data. Furthermore, the `sql read` function does not require you to set database preferences.

For maximum memory savings, you can import and access data in one step 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, using the `select` function affords maximum memory savings.

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

Database Data Type	MATLAB Data Type
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. 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.

Data Import in Two Steps

You can import data in two steps using the `exec` and `fetch` functions. Running `exec` executes the SQL query.

- If you are using the native ODBC interface, then `exec` moves the results of the query from the database server into the main computer memory, or RAM.
- If you are using a JDBC driver, then `exec` moves the results into the Java heap.

The `fetch` function imports all numeric values from the database, and then stores them in a double-precision array in the MATLAB workspace. Double-precision arrays consume more memory than commensurate arrays containing integer data types. To save memory, use the `select` function to import integer values from the database into the MATLAB workspace as the matching integer data types.

Large Data Import Using Row Limits

To manage memory by limiting the number of rows that are imported from an executed query, use one of these options:

- Manage RAM or Java heap memory depending on the driver:
 - Use the input argument `rowlimit` in the `exec` function.

- Modify the query by adding LIMIT or a similar SQL syntax at the end of the SQL statement. For example, this SQL statement assumes a MySQL database connection.

```
sqlquery = 'SELECT * FROM productTable LIMIT 5';
```

- Manage MATLAB memory:
 - Use the input argument `rowlimit` in the `fetch` function.
 - For a SQL script, use the name-value pair argument `'RowInc'` in the `runsqlscript` function.
 - For the MATLAB interface to SQLite, use the input argument `rowlimit` in the `fetch` function.

See Also

`exec` | `fetch` | `runsqlscript` | `select` | `sqlread`

More About

- “Data Import Using Database Explorer App or Command Line” on page 2-145
- “Import Data from Databases into MATLAB” on page 5-3
- “Using Scrollable Cursors” on page 5-53
- “Working with Large Data Sets” on page 2-150

Display Information About Imported Data

This example shows how to import data and display information about the imported data using a cursor object. Here, metadata refers to the information about the cursor object that contains the imported data after running `exec`.

Create Database Connection and Import Data

Create the database connection `conn` using the `dbdemo` data source. `dbdemo` contains the table `productTable` with the column `productDescription`.

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

Create the cursor object `curs` by selecting data in `productDescription` from `productTable` using `conn`. `sqlquery` contains the SQL `SELECT` statement for this query.

```
sqlquery = 'SELECT productDescription FROM productTable';
```

```
curs = exec(conn,sqlquery);
```

Determine if the cursor object `curs` is open.

```
o = isopen(curs)
```

```
o =
```

```
1
```

`o` returns as the scalar `1` that denotes the cursor object is open.

Import the first 10 rows of product description data using `curs`.

```
curs = fetch(curs,10);
```

`fetch` stores the imported data in the cursor object property `curs.Data`.

Retrieve Number of Rows in Imported Data

Retrieve the number of rows `numrows` using `curs`.

```
numrows = rows(curs)
```

```
numrows =  
    10
```

Retrieve Number of Columns in Imported Data

Retrieve the number of columns `numcols` using `curs`.

```
numcols = cols(curs)  
numcols =  
    1
```

Retrieve Column Name in Imported Data

Retrieve the column name `colname` using `curs`.

```
colname = columnnames(curs)  
colname =  
    'productDescription'
```

Retrieve Column Width in Imported Data

Retrieve the column width `colsize`, or size of the field, for the first column using `curs`.

```
colsize = width(curs,1)  
colsize =  
    50
```

Display Attributes in Imported Data

Display the attributes for the product description column using `curs`.

```
attributes = attr(curs)  
attributes =  
    fieldName: 'productDescription'  
    typeName: 'VARCHAR'  
    typeValue: 12  
    columnWidth: 50
```

```
precision: []
scale: []
currency: 'false'
readOnly: 'false'
nullable: 'true'
Message: []
```

Close cursor Object

After you finish working with the `cursor` object, close it.

```
close(curs)
```

See Also

`attr` | `cols` | `columnnames` | `database` | `fetch` | `rows` | `setdbprefs` | `width`

Related Examples

- “Import Data from Databases into MATLAB” on page 5-3

External Websites

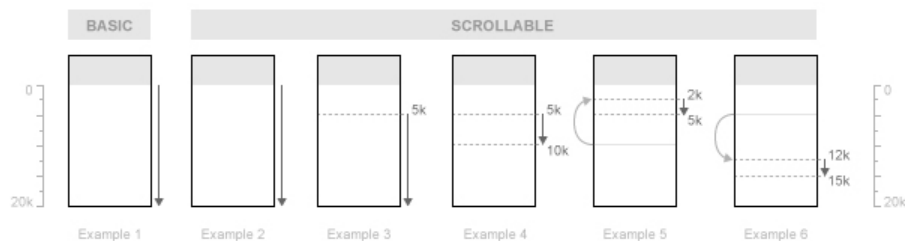
- [SQL Tutorial](#)

Using Scrollable Cursors

Scrollable Cursors

A basic cursor lets you fetch the data in your SQL query sequentially. With a scrollable cursor, you can fetch data sequentially or scroll up or down in the data without rerunning the query. The cursor changes position based on an absolute or relative offset value. Scrolling within the data offers advantages when you are working with a large data set.

This diagram shows the differences between the basic and scrollable cursors. Each example in the diagram shows fetching data in the same table that contains 20,000 records.



As shown in Example 1, the basic cursor lets you fetch data sequentially. As shown by Examples 2 through 6, the scrollable cursor lets you do this and fetch data from an absolute or relative cursor position. Examples 3 and 4 use an absolute position offset and Examples 5 and 6 use a relative position offset.

Scrollable cursors let you fetch data from a specific position. Example 3 fetches all records starting from the absolute cursor position of 5000. Example 4 fetches 5000 records starting from the absolute cursor position of 5000.

Further, scrollable cursors let you fetch data relative to your current cursor position. Assuming your current cursor position is 10,000, Example 5 fetches 3000 records using a relative cursor position offset of -8000. A negative position offset moves the scrollable cursor backwards in the data set. The `fetch` function adds -8000 to the current cursor position of 10,000 to start fetching data from 2000. Assuming your cursor stays at the position of 5000 after fetching data in Example 5, Example 6 fetches 3000 records using a relative cursor position offset of 7000. A positive position offset moves the scrollable cursor forward in the data set. The `fetch` function adds 7000 to the current cursor position of 5000 to start fetching data from 12,000.

To use a scrollable cursor, first you need to create it by using the `exec` function. This code creates a scrollable cursor object `curs` using a database connection `conn` and an SQL query `sqlquery` .

```
curs = exec(conn,sqlquery, 'CursorType', 'scrollable');
```

Then, you can use `fetch` to retrieve data in the cursor with an offset. The offset lets you retrieve data starting from the middle of the data set. You cannot retrieve data with an offset using a basic cursor object. As you continue to fetch, the position of the cursor changes. You can enter `curs.Position` to see the current position of the cursor object `curs` , or you can use `get` .

The database driver for your database determines if scrollable cursor functionality is available. Consult your database documentation to ensure your database driver supports scrollable cursors.

Differences Between Native ODBC and JDBC Scrollable Cursors

Native ODBC and JDBC drivers implement scrollable cursor functionality differently. Further, database drivers implement scrollable cursor functionality differently. Both tables illustrate the differences in scrollable cursor behavior across drivers. The rows depict examples of using a scrollable cursor with native ODBC and JDBC connections. For each row, the full data set has 15 records. Each table row shows the values for the input arguments in a specific call of the `fetch` function. The column descriptions show that:

- The Initial Scrollable Cursor Position column captures the value of the cursor position before calling `fetch` .
- The Row Limit column shows values for the `rowlimit` input argument in `fetch` .
- The Scrollable Cursor Position Type column specifies the name in the name-value pair argument for the cursor position offset.
- The Offset column specifies the value in the name-value pair argument for the cursor position offset.
- The Ending Scrollable Cursor Position column captures the value of the cursor position after calling `fetch` .
- The `fetch` Action column describes the rows of data to retrieve based on the specified input arguments.

For example, this code demonstrates the syntax for calling `fetch` shown in the second row of either table.

```
curs = fetch(curs,2,'AbsolutePosition',1);
```

Native ODBC

Initial Scrollable Cursor Position	Row Limit	Scrollable Cursor Position Type	Offset	Ending Scrollable Cursor Position	fetch Action
Any	Not specified	'AbsolutePosition'	1	After the result set	Retrieves all rows in the cursor starting from the first row in the data set
Any	2	'AbsolutePosition'	1	1	Retrieves two rows in the cursor starting from the first row in the data set
Any	2	'AbsolutePosition'	5	5	Retrieves two rows in the cursor starting from the fifth row in the data set
Any	3	'AbsolutePosition'	-5	11	Retrieves three rows in the cursor starting from the fifth row from the end of the data set
Before result set	Not specified	'RelativePosition'	1	After the result set	Retrieves all rows in the cursor starting from the first row in the data set

Initial Scrollable Cursor Position	Row Limit	Scrollable Cursor Position Type	Offset	Ending Scrollable Cursor Position	fetch Action
Before result set	Any	'RelativePosition'	Any	Varies	Retrieving with a relative position that starts before the result set causes behavior to vary based on the driver
5	2	'RelativePosition'	5	10	Retrieves two rows in the cursor starting from the tenth row in the data set
11	3	'RelativePosition'	-5	6	Retrieves three rows in the cursor starting from the sixth row in the data set

JDBC

Initial Scrollable Cursor Position	Row Limit	Scrollable Cursor Position Type	Offset	Ending Scrollable Cursor Position	fetch Action
Any	Not specified	'AbsolutePosition'	1	0	Retrieves all rows in the cursor starting from the first row in the data set
Any	2	'AbsolutePosition'	1	2	Retrieves two rows in the cursor starting from the first row in the data set
Any	2	'AbsolutePosition'	5	6	Retrieves two rows in the cursor starting from the fifth row in the data set
Any	3	'AbsolutePosition'	-5	13	Retrieves three rows in the cursor starting with the fifth row from the end of the data set. This assumes there are 15 records in the data set.

Initial Scrollable Cursor Position	Row Limit	Scrollable Cursor Position Type	Offset	Ending Scrollable Cursor Position	fetch Action
0	Not specified	'RelativePosition'	1	0	Retrieves all rows in the cursor starting from the first row in the data set
0	2	'RelativePosition'	1	2	Retrieves the first two rows in the data set
5	2	'RelativePosition'	5	11	Retrieves two rows in the data set starting from five rows from the initial position of five, which is nine
11	3	'RelativePosition'	-5	8	Retrieves three rows in the cursor starting from five rows before the eleventh row in the data set

See Also

exec | fetch | get

Related Examples

- “Import Data from Databases into MATLAB” on page 5-3

More About

- “Import Data Using Scrollable Cursor with Relative Position Offset” on page 5-61

Import Data Using Scrollable Cursor with Relative Position Offset

This example shows how to use a scrollable cursor to import data using both absolute and relative position offsets. This example assumes you are connecting to a MySQL database that contains a table called `productTable`. This table contains 15 records, where each record represents one product. The scrollable cursor functionality behaves differently depending on your database driver. For details about the scrollable cursor functionality in your database, consult your database documentation.

Connect to Database

Connect to the MySQL database using an ODBC driver. This code assumes you are connecting to a data source named `MySQL` with user name `username` and password `pwd`.

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

Create Scrollable Cursor

Select all products from the `productTable` table and sort them in ascending order by product number. Create a scrollable cursor using the name-value pair argument `'CursorType'`.

```
sqlquery = 'SELECT * FROM productTable ORDER BY productNumber';
curs = exec(conn, sqlquery, 'CursorType', 'scrollable');
```

Retrieve Data Using Absolute Position Offset

Import the data for two products in the middle of the data set. Use the row limit 2 to import data for two products. Use the absolute position offset 5 to import data starting from the fifth product in the data set.

```
curs = fetch(curs, 2, 'AbsolutePosition', 5);
```

Display the data for the two products.

```
curs.Data
```

```
ans =
```

```

[5]    [400455]    [1005]    [3]    'Tin Soldier'
[6]    [400876]    [1004]    [8]    'Sail Boat'
```

The columns in `curs.Data` are:

- Product number
- Stock number
- Supplier number
- Unit cost
- Product description

Display the position of the cursor.

```
curs.Position
```

```
ans =
```

```
5
```

The position of the cursor stays at the absolute position offset 5.

Retrieve Data Using Relative Position Offset

Import the data for three products in the data set using the relative position offset 5. A scrollable cursor adds the current position offset 5 to the specified relative position offset 5. The scrollable cursor advances to cursor position 10 and imports data.

```
curs = fetch(curs,3,'RelativePosition',5);
```

Display the data for the three products.

```
curs.Data
```

```
ans =
```

```
    [10]    [888652]    [1006]    [24]    'Teddy Bear'  
    [11]    [408143]    [1004]    [11]    'Convertible'  
    [12]    [210456]    [1010]    [22]    'Hugsy'
```

Display the position of the cursor.

```
curs.Position
```

```
ans =
```

```
10
```

Close cursor Object

After you finish working with the cursor object, close it.

```
close(curs)
```

See Also

`close` | `database` | `exec` | `fetch`

Related Examples

- “Import Data from Databases into MATLAB” on page 5-3

More About

- “Using Scrollable Cursors” on page 5-53

External Websites

- [SQL Tutorial](#)

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.

Create DatabaseDatastore Object

Using a JDBC driver, create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. Here, the code assumes that you are connecting to database `toy_store`, database server `dbtb04`, and port number 54317.

```
conn = database('toy_store', '', 'Vendor', 'Microsoft SQL Server', ...
    'Server', 'dbtb04', 'PortNumber', 54317, 'AuthType', 'Windows');
```

Create a `DatabaseDatastore` object using the database connection and SQL query. This SQL query retrieves all data from the 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 `sqlquery`.

```
preview(dbds)
```

```
ans =
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRS
1990	9	22	6	1801	1750	2005	193
1990	9	11	2	908	910	1613	155
1990	9	2	7	NaN	1805	NaN	190
1990	9	29	6	1434	1435	1615	163

1990	9	3	1	925	755	1258	114
1990	9	22	6	900	900	1241	122
1990	9	20	4	1338	1335	1853	190
1990	9	3	1	710	711	837	84

Read Data in DatabaseDatastore Object

Read the first 10 records.

```
dbds.ReadSize = 10;
```

```
read(dbds)
```

```
ans =
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRS
1987	10	28	3	1140	1140	1212	122
1987	10	9	5	1155	1155	1250	130
1987	10	22	4	715	715	807	80
1987	10	16	5	1553	1555	1641	164
1987	10	30	5	1821	1815	1956	195
1987	10	12	1	1300	1300	1529	152
1987	10	7	3	810	810	904	90
1987	10	19	1	733	735	827	83
1987	10	15	4	828	830	916	92
1987	10	4	7	1750	1735	1837	183

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 =
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRS
1987	10	16	5	959	1000	1212	123
1987	10	17	6	2020	2020	2100	209
1987	10	6	2	1132	1135	1426	143
1987	10	24	6	944	945	1211	123
1987	10	18	7	833	835	1003	103
1987	10	26	1	2356	2355	730	73
1987	10	29	4	1056	1055	1208	123
1987	10	1	4	2304	2255	2340	233
1987	10	30	5	1329	1329	1434	143
1987	10	3	6	1040	1040	1125	113

ans =

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRS
1987	10	23	5	1855	1855	2158	220
1987	10	30	5	1055	1055	1302	133
1987	10	28	3	NaN	1850	NaN	209
1987	10	26	1	1600	1600	1649	164
1987	10	6	2	745	745	833	83
1987	10	31	6	1350	1350	1612	163
1987	10	12	1	1253	1200	1359	133
1987	10	19	1	650	645	852	83
1987	10	10	6	1640	1640	1712	172
1987	10	2	5	2030	2030	2127	213

Reset DatabaseDatastore Object

Reset the DatabaseDatastore object to the state where no data has been read from it. Resetting allows rereading 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 =
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRS
1987	10	28	3	1140	1140	1212	12
1987	10	9	5	1155	1155	1250	13
1987	10	22	4	715	715	807	8

Close DatabaseDatastore Object and Database Connection

```
close(dbds)
```

See Also

[close](#) | [database](#) | [databaseDatastore](#) | [hasdata](#) | [preview](#) | [read](#) | [readall](#) | [reset](#)

Related Examples

- “Analyze Large Data in Database Using Tall Arrays”
- “Analyze Large Data in Database Using MapReduce”
- Building Effective Algorithms with MapReduce (MATLAB)

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 a SQLite database file. You do not need to install a database or driver. For details about the MATLAB® interface to SQLite, see “Working with MATLAB Interface to SQLite” on page 2-6. For more functionality, connect to the SQLite database file using the JDBC driver. For details, see “Configuring Driver and Data Source” on page 2-15.

To access the code for this example, enter `edit SQLiteWorkflow.m`.

Create SQLite Connection

Create a 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 `exec`. Clear the MATLAB® workspace variables.

```
createInventoryTable = ['create table inventoryTable ' ...  
    '(productNumber NUMERIC, Quantity NUMERIC, ' ...  
    'Price NUMERIC, inventoryDate VARCHAR)'];  
exec(conn, createInventoryTable)  
  
createSuppliers = ['create table suppliers ' ...  
    '(SupplierNumber NUMERIC, SupplierName varchar(50), ' ...  
    'City varchar(20), Country varchar(20), ' ...  
    'FaxNumber varchar(20))'];  
exec(conn, createSuppliers)  
  
createSalesVolume = ['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)'];
```



```

exec(conn,createSalesVolume)

createProductTable = ['create table productTable ' ...
    '(productNumber NUMERIC, stockNumber NUMERIC, ' ...
    'supplierNumber NUMERIC, unitCost NUMERIC, ' ...
    'productDescription varchar(20))'];
exec(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 `insert`. Clear the MATLAB® workspace variables.

```

load('sqliteworkflowdata.mat')

insert(conn,'inventoryTable', ...
    {'productNumber','Quantity','Price','inventoryDate'},CinvTable)

insert(conn,'suppliers', ...
    {'SupplierNumber','SupplierName','City','Country','FaxNumber'}, ...
    Csuppliers)

insert(conn,'salesVolume', ...
    {'StockNumber','January','February','March','April','May','June', ...
    'July','August','September','October','November','December'}, ...
    CsalesVol)

insert(conn,'productTable', ...
    {'productNumber','stockNumber','supplierNumber','unitCost', ...
    'productDescription'},CprodTable)

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.

```
inventoryTable_data(1:3,:)
```

```
ans = 3x4 cell array
    {[1]}    {[1700]}    {[14.5000]}    {'9/23/2014 9:38...'}
    {[2]}    {[1200]}    {[ 9.3000]}    {'7/8/2014 10:50...'}
    {[3]}    {[ 356]}    {[17.2000]}    {'5/14/2014 7:14...'}

```

```
suppliers_data(1:3,:)
```

```
ans = 3x5 cell array
Columns 1 through 4

    {[1001]}    {'Wonder Products'}    {'New York'}    {'United States' }
    {[1002]}    {'Terrific Toys' }    {'London' }    {'United Kingdom'}
    {[1003]}    {'Wacky Widgets' }    {'Adelaide'}    {'Australia' }

Column 5

    {'212 435 1617' }
    {'44 456 9345' }
    {'618 8490 2211' }
```

```
salesVolume_data(1:3,:)
```

```
ans = 3x13 cell array
      Columns 1 through 6
```

```
    {[125970]}    {[1400]}    {[1100]}    {[ 981]}    {[ 882]}    {[794]}
    {[212569]}    {[2400]}    {[1721]}    {[1414]}    {[1191]}    {[983]}
    {[389123]}    {[1800]}    {[1200]}    {[ 890]}    {[ 670]}    {[550]}
```

```
      Columns 7 through 12
```

```
    {[752]}    {[654]}    {[773]}    {[809]}    {[980]}    {[3045]}
    {[825]}    {[731]}    {[653]}    {[723]}    {[790]}    {[1400]}
    {[450]}    {[400]}    {[410]}    {[402]}    {[450]}    {[1200]}
```

```
      Column 13
```

```
    {[19000]}
    {[ 5000]}
    {[16000]}
```

```
productTable_data(1:3,:)
```

```
ans = 3x5 cell array
```

```
    {[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

close | exec | fetch | insert | sqlite

More About

- “Access Relational Database Data in MATLAB” on page 2-3
- “Working with MATLAB Interface to SQLite” on page 2-6
- “Configuring Driver and Data Source” on page 2-15

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 Specify the data return format to be a cell array.

```
setdbprefs('DataReturnFormat','cellarray')
```

- 3 Import the InvoiceNumber and Receipt columns of data from Invoice.

```
sqlquery = 'SELECT InvoiceNumber,Receipt FROM Invoice';
curs = exec(conn,sqlquery);
curs = fetch(curs);
```

- 4 View the imported data.

```
curs.Data
```

```
ans =
```

```

[ 2101]    [1948410x1 int8]
[ 3546]    [2059994x1 int8]
[ 33116]   [ 487034x1 int8]
[ 34155]    [2059994x1 int8]

```

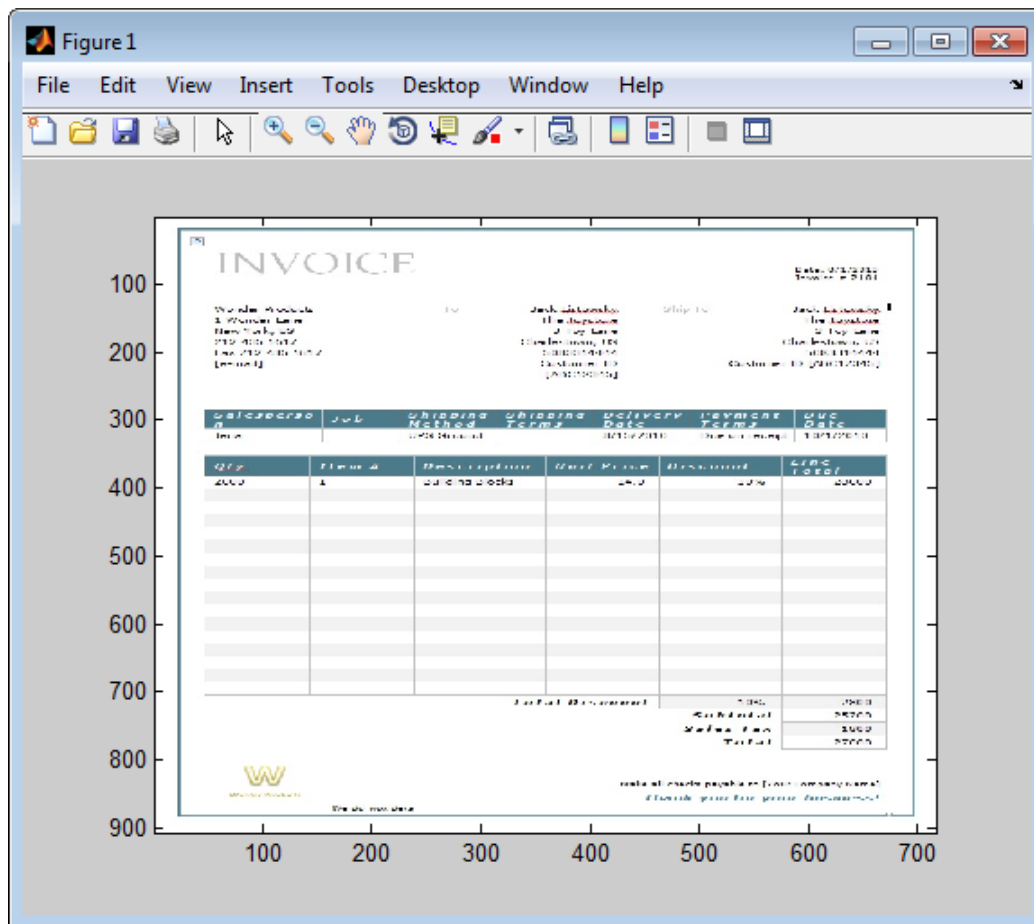
```
[ 34267]      [2454554x1 int8]
[ 37197]      [1926362x1 int8]
[ 37281]      [2403674x1 int8]
[ 41011]      [1920474x1 int8]
[ 61178]      [2378330x1 int8]
[ 62145]      [ 492314x1 int8]
[456789]      []
[987654]      []
```

- 5** Assign the image element you want to the variable `receipt`.

```
receipt = curs.Data{1,2};
```

- 6** 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 | exec | fetch | setdbprefs

Related Examples

- “Import Data from Databases into MATLAB” on page 5-3

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 a 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 =  
  
    11×1 table  
  
    Paid  
    ———  
  
    false  
    true  
    true  
    false  
    true  
    true  
    false
```

```
true  
false  
true  
false
```

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 =
```

```
5
```

Close Database Connection

```
close(conn)
```

See Also

`close` | `database` | `select`

Related Examples

- “Import Data from Databases into MATLAB” on page 5-3
- “Export Data to New Record in Database” on page 5-21

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
```

```
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

`cell2table` | `detectImportOptions` | `readtable` | `setvartype` | `sqlwrite` | `table2cell`

More About

- “Insert Data into New Database Table Using Insert Functionality” on page 5-82
- “Insert Data into Database Table” on page 5-88
- “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)'];
exec(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

`cell2table` | `exec` | `readtable` | `sqlwrite` | `table2cell`

More About

- “Append Data to Existing Database Table Using Insert Functionality” on page 5-80
- “Insert Data into Database Table” on page 5-88
- “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 4-6. Or, you can create and run an SQL script using the `runsqlscript` function.

See Also

`sqlinnerjoin` | `sqlouterjoin`

More About

- “Join Tables Using Database Explorer App” on page 4-6
- “Fetching Data Common Errors” on page 3-4

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 SQL script file named `salesvolume.sql` by using the `runsqlscript` function. `results` is a cursor object array with the data returned from running the SQL query in the SQL script file.

```
results = runsqlscript(conn, 'salesvolume.sql');
```

Display the first three rows in the `Data` table. Access this table as a property of the cursor object by using dot notation.

```
head(results(1).Data,3)
```

```
ans =
```

```
3×13 table
```

January	February	March	April	May	June	July	August	September
5000	3500	2800	2300	1700	1400	1000	900	1600
2400	1721	1414	1191	983	825	731	653	723
1200	900	800	500	399	345	300	175	760

Close Database Connection

```
close(conn)
```

See Also

[close](#) | [database](#) | [sqlread](#)

More About

- “Import Data from Databases into MATLAB” on page 5-3
- “Data Import Approaches and Memory Management” on page 5-46

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`. 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 `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 =
```

```
1×3 table
```

Month	salesTotal	Revenue
'March'	14606	0

Close Database Connection

```
close(conn)
```

See Also

[close](#) | [database](#) | [sqlread](#) | [sqlwrite](#)

More About

- “Export Data Using Bulk Insert” on page 5-32
- “Inserting Data Using Command Line” on page 2-148

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 Microsoft® SQL Server® database that contains a table named `productTable`.

Connect to 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 Catalogs and Schemas

Display the catalogs in the database by using the `Catalogs` property of the `connection` object.

```
conn.Catalogs
```

```
ans =  
  
1×4 cell array  
  
{'master'} {'msdb'} {'tempdb'} {'toy_store'}
```

Display the first three schemas in the database by using the `Schemas` property of the `connection` object.

```
conn.Schemas{1:3}
```

```
ans =
```

```
    'dbo'
```

```
ans =
```

```
    'guest'
```

```
ans =
```

```
    'INFORMATION_SCHEMA'
```

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
'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'

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...'}  
ans =
```

Close Database Connection

```
close(conn)
```

See Also

[close](#) | [database](#) | [sqlfind](#) | [sqlread](#)

More About

- “Import Data from Database Table Using `sqlread` Function” on page 5-85

Neo4j Topics

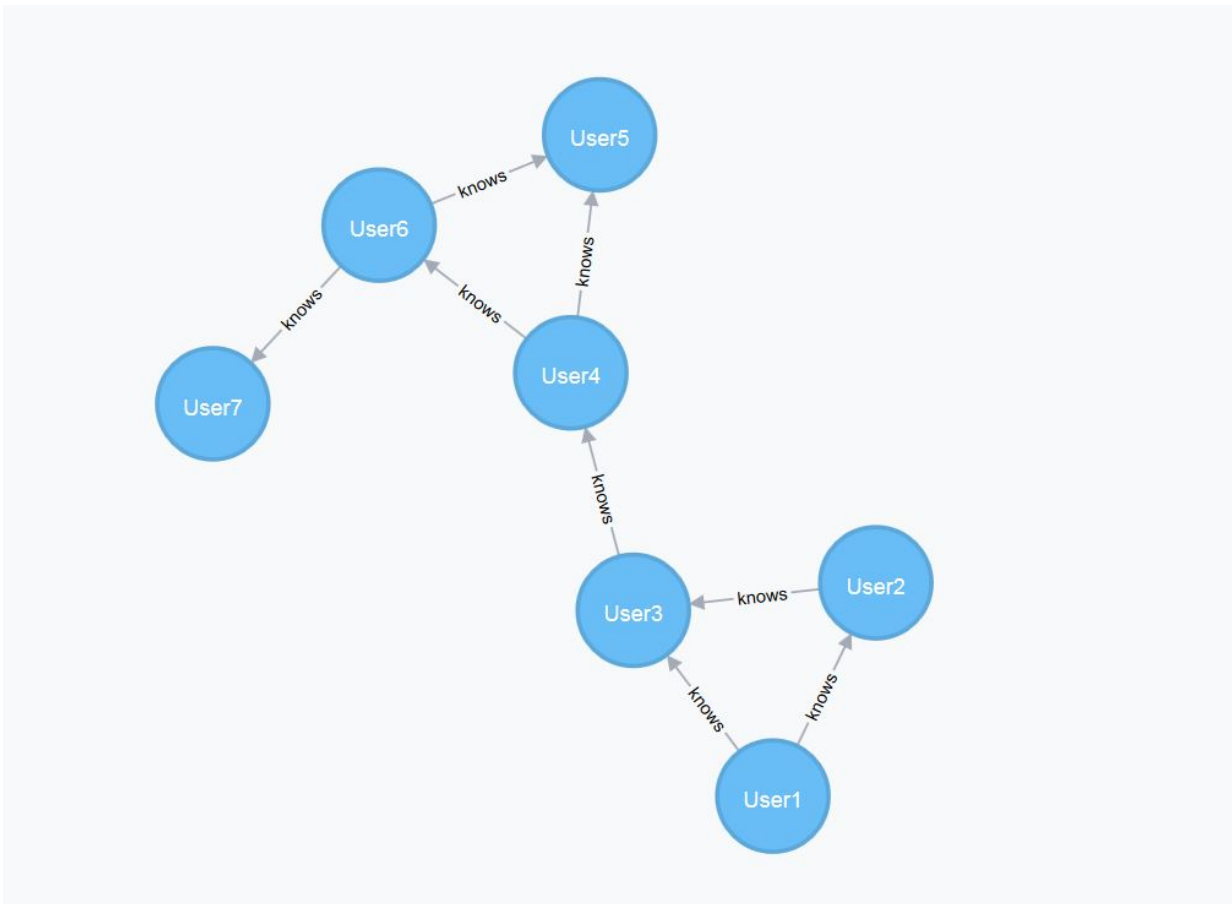
- “Explore Graph Database Structure” on page 6-2
- “Working with MATLAB Interface to Neo4j” on page 6-9
- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12
- “Update Friend Information in Social Neighborhood” on page 6-15
- “Add and Query Group of Colleagues in Social Neighborhood” on page 6-20
- “MATLAB Interface to Neo4j Error Messages” on page 6-28

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 “Working with MATLAB Interface to Neo4j” on page 6-9.

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`.

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`.

```
neo4jconn.Message
```

```
ans =
```

```
  []
```

The blank Message property indicates a successful connection.

Explore Structure of Entire Graph

Find all the node labels in the Neo4j database using the Neo4j connection object neo4jconn.

```
nlabels = nodeLabels(neo4jconn)
```

```
nlabels =
```

```
  1×1 cell array
```

```
  {'Person'}
```

Find all the relationship types in the Neo4j database.

```
reltypes = relationTypes(neo4jconn)
```

```
reltypes =
```

```
  1×1 cell array
```

```
  {'knows'}
```

Find the property keys in the Neo4j database.

```
propkeys = propertyKeys(neo4jconn)
```

```
propkeys =
```

```
  12×1 cell array
```

```

{'Address'    }
{'Title'     }
{'property'  }
{'Location'  }
{'Description'}
{'name'      }
{'Name'      }
{'Date'     }
{'Project'   }
{'StartDate' }
{'Weight'    }
{'title'    }

```

Search for Nodes

Search for all the nodes with the node label Person.

```

nlabel = 'Person';
nodesinfo = searchNode(neo4jconn,nlabel)

```

nodesinfo =

7×3 table

	NodeLabels	NodeData	NodeObject
0	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
1	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
2	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
3	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
4	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
5	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
9	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]

nodesinfo contains node labels, node data, and the Neo4jNode objects for each matched node.

Search for the node with the node identifier 2.

```
nodeid = 2;
nodeinfo = searchNodeByID(neo4jconn,nodeid)

nodeinfo =
    Neo4jNode with properties:
        NodeID: 2
        NodeData: [1x1 struct]
        NodeLabels: 'Person'
```

nodeinfo contains the node identifier, node data, and node labels for the node with node identifier 2.

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`.

```
reinfo = searchRelation(neo4jconn,nodeinfo,'in')

reinfo =
    struct with fields:
        Origin: 2
        Nodes: [2x3 table]
        Relations: [1x5 table]
```

`reinfo` contains data about the start and end nodes and all matched relationships from the origin node.

Retrieve Entire Graph

Retrieve the entire graph using node labels `nlabels`.

```
graphinfo = searchGraph(neo4jconn,nlabels)

graphinfo =
    struct with fields:
        Nodes: [7x3 table]
```

Relations: [8×5 table]

graphinfo contains node data for all start and end nodes for each matched relationship. graphinfo also contains relationship data for each matched relationship.

See Also

neo4j | nodeDegree | nodeLabels | nodeRelationTypes | propertyKeys | relationTypes | searchNode | searchNodeByID | searchRelation | searchRelationByID

More About

- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12
- “Working with MATLAB Interface to Neo4j” on page 6-9

External Websites

- [Neo4j Documentation](#)

Working with MATLAB Interface to Neo4j

The MATLAB interface to Neo4j enables you to:

- Create a Neo4j database connection.
- 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 the nodes and relationships in the graph database.
- 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.

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 of different relationship types.

For details about graphs, see “Directed and Undirected Graphs” (MATLAB). For details about the Neo4j database, see [Why Graph Databases?](#)

MATLAB Interface to Neo4j 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 with any of these functions:

- `nodeLabels`
- `propertyKeys`
- `relationTypes`
- `searchGraph`

Or, conduct a targeted search in the graph database with any of these functions:

- `searchNode`
- `searchNodeByID`
- `searchRelation`
- `searchRelationByID`
- `nodeDegree`
- `nodeRelationTypes`

3 Update the graph database.

Create nodes and relationships with these functions:

- `createNode`
- `createRelation`

Update nodes and relationships with these functions:

- `updateNode`
- `updateRelation`

Delete nodes and relationships with these functions:

- `deleteNode`
- `deleteRelation`

4 Export a directed graph from MATLAB into a Neo4j database 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” (MATLAB).

Or, if you know the Cypher query language, you can execute a Cypher query using `executeCypher`. For details, see Cypher Query Language.

See Also

digraph | neo4j | neo4jStruct2Digraph

More About

- “Find Friends of Friends in Social Neighborhood”
- “Visualize Breadth-First and Depth-First Search” (MATLAB)
- “Directed and Undirected Graphs” (MATLAB)
- “Update Friend Information in Social Neighborhood” on page 6-15
- “Add and Query Group of Colleagues in Social Neighborhood” on page 6-20
- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12
- “MATLAB Interface to Neo4j Error Messages” on page 6-28

External Websites

- Neo4j Documentation
- Cypher Query Language

Searching Graph Database Using MATLAB Interface to Neo4j

Search the Neo4j graph database using functions provided by the MATLAB interface to Neo4j. You can explore the graph data or perform graph network analysis using MATLAB directed graphs.

MATLAB Interface to Neo4j Search Functions

Search graph data in the Neo4j graph database using different parts of the graph:

- To search for nodes, use one of two functions. 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 the 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`.

```
nlabel = {'Person'};
```

```
graphinfo = searchGraph(neo4jconn,nlabel);
```

- 2 Convert the output structure `graphinfo` into a `digraph` object `G` using `neo4jStruct2Digraph`.

```
G = neo4jStruct2Digraph(graphinfo);
```

- 3 Perform graph network analysis using `G`. For details, see “Directed and Undirected Graphs” (MATLAB).

For example, determine the shortest path between nodes using `distances`.

```
d = distances(G);
```

Or, explore the graph data by 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.

```
relinfo = searchRelation(neo4jconn,nodeinfo,'in');
```

- 3 Convert the output structure `relinfo` into a `digraph` object `G` by using `neo4jStruct2Digraph`.

```
G = neo4jStruct2Digraph(relinfo);
```

- 4 Perform graph network analysis using the `digraph` object `G`. For details, see “Directed and Undirected Graphs” (MATLAB).

For example, determine the shortest path between nodes using `distances`.

```
d = distances(G);
```

Or, explore the graph data by accessing the output structures `nodeinfo` and `relinfo`.

See Also

`nodeDegree` | `searchGraph` | `searchNode` | `searchNodeByID` | `searchRelation`

More About

- “Explore Graph Database Structure” on page 6-2
- “Find Shortest Path Between People in Social Neighborhood”
- “Working with MATLAB Interface to Neo4j” on page 6-9
- “MATLAB Interface to Neo4j Error Messages” on page 6-28

External Websites

- [Neo4j Documentation](#)

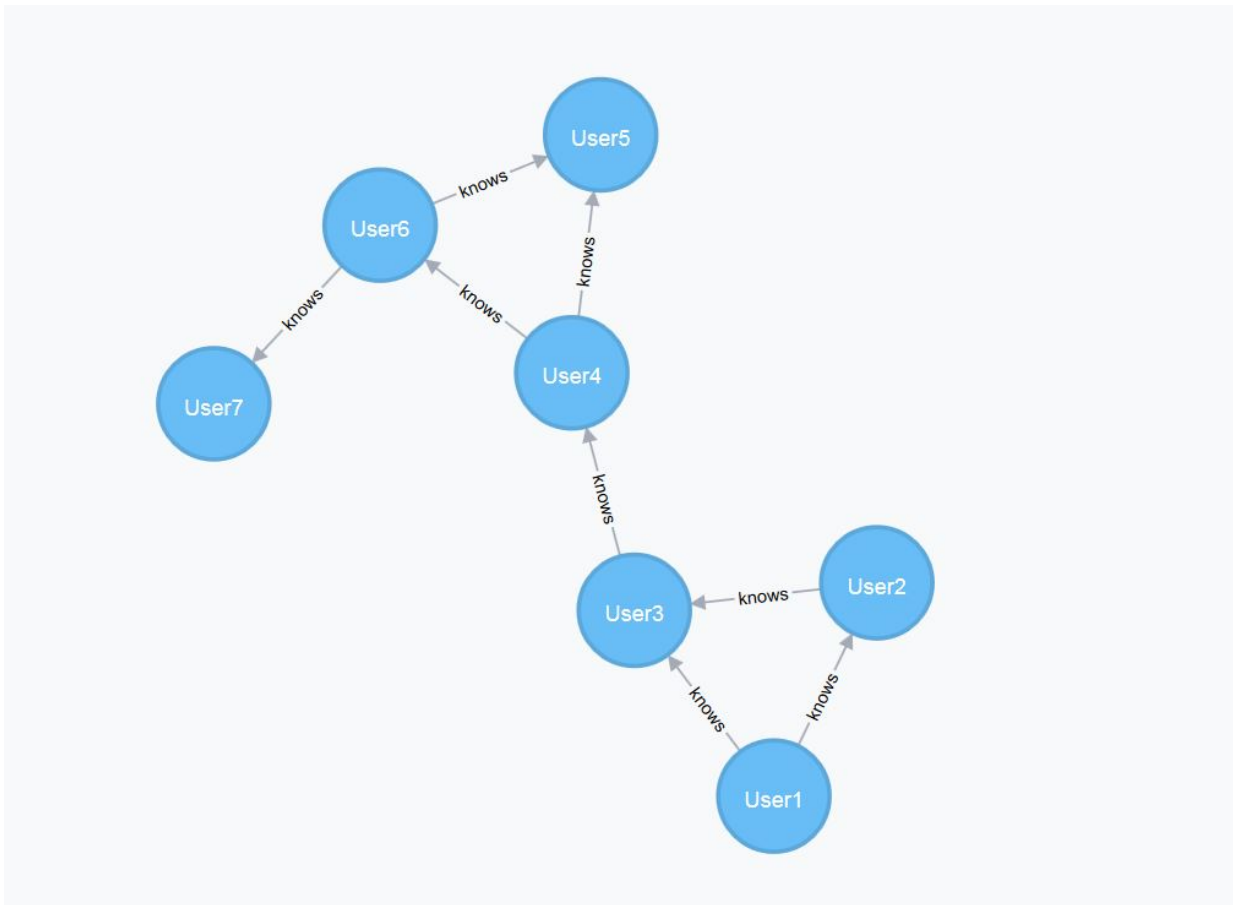
Update Friend Information in Social Neighborhood

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 “Working with MATLAB Interface to Neo4j” on page 6-9.

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 values `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 `neo4j conn`. The blank `Message` property indicates a successful connection.

```
neo4j conn.Message
```

```
ans =
```

```
 []
```

Add Two Friends to Social Neighborhood

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';
user8 = createNode(neo4j conn, 'Labels', label);
user9 = createNode(neo4j conn, 'Labels', label);
```

Search for the node with the node label `Person` and the property key `name` set to the value `User7` by using the Neo4j database connection.

```
nlabel = 'Person';
user7 = searchNode(neo4j conn, 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`

The relationship `relationinfo` is a table that contains the relationship and node information.

```
startnode = [user8, user8];
endnode = [user7, user9];
relationtype = {'works with', 'studies with'};
relationinfo = createRelation(neo4j conn, 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 = updateNode(neo4jconn,user8,'Properties',properties);
```

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 = updateRelation(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)
```

See Also

Neo4jNode | createNode | createRelation | deleteNode | deleteRelation | neo4j | updateNode | updateRelation

More About

- “Find Shortest Path Between People in Social Neighborhood”
- “Find Friends of Friends in Social Neighborhood”

- “Working with MATLAB Interface to Neo4j” on page 6-9

External Websites

- Neo4j Documentation

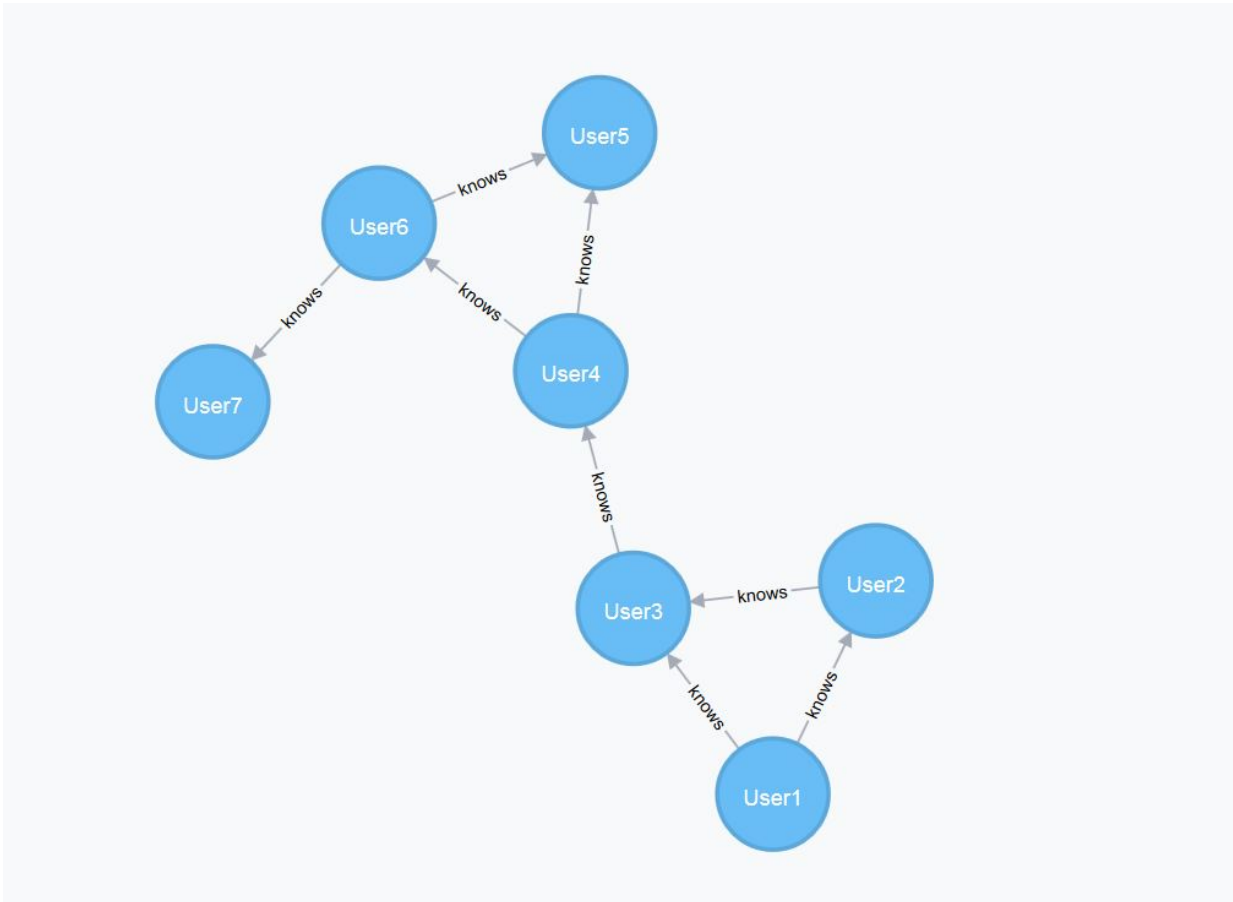
Add and Query Group of Colleagues in Social Neighborhood

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 “Working with MATLAB Interface to Neo4j” on page 6-9.

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 values `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 `neo4j conn`. The blank `Message` property indicates a successful connection.

```
neo4j conn.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.

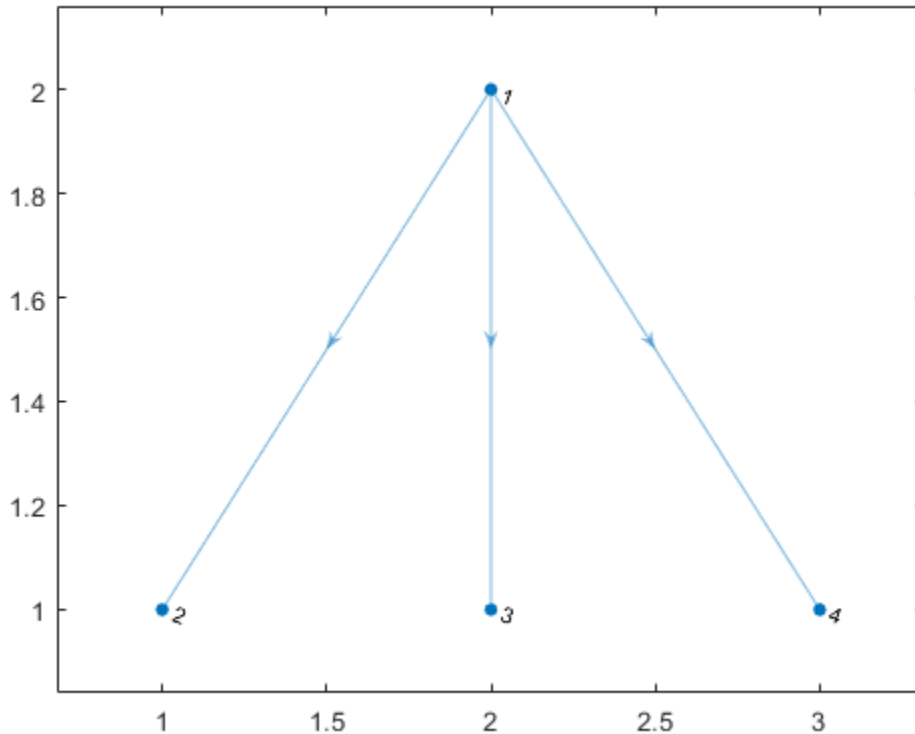
```
s = [1 1 1];  
t = [2 3 4];  
G = digraph(s,t);
```

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  Relati  
-----  
14         14      'works with'      18      [1x1 struct]  [1x1 database.ne  
13         14      'works with'      17      [1x1 struct]  [1x1 database.ne  
12         14      'works with'      15      [1x1 struct]  [1x1 database.ne
```

Relations is a table that contains these variables:

- Start node identifiers
- Relationship types
- End node identifiers
- Relationship properties
- Neo4jRelation objects

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.

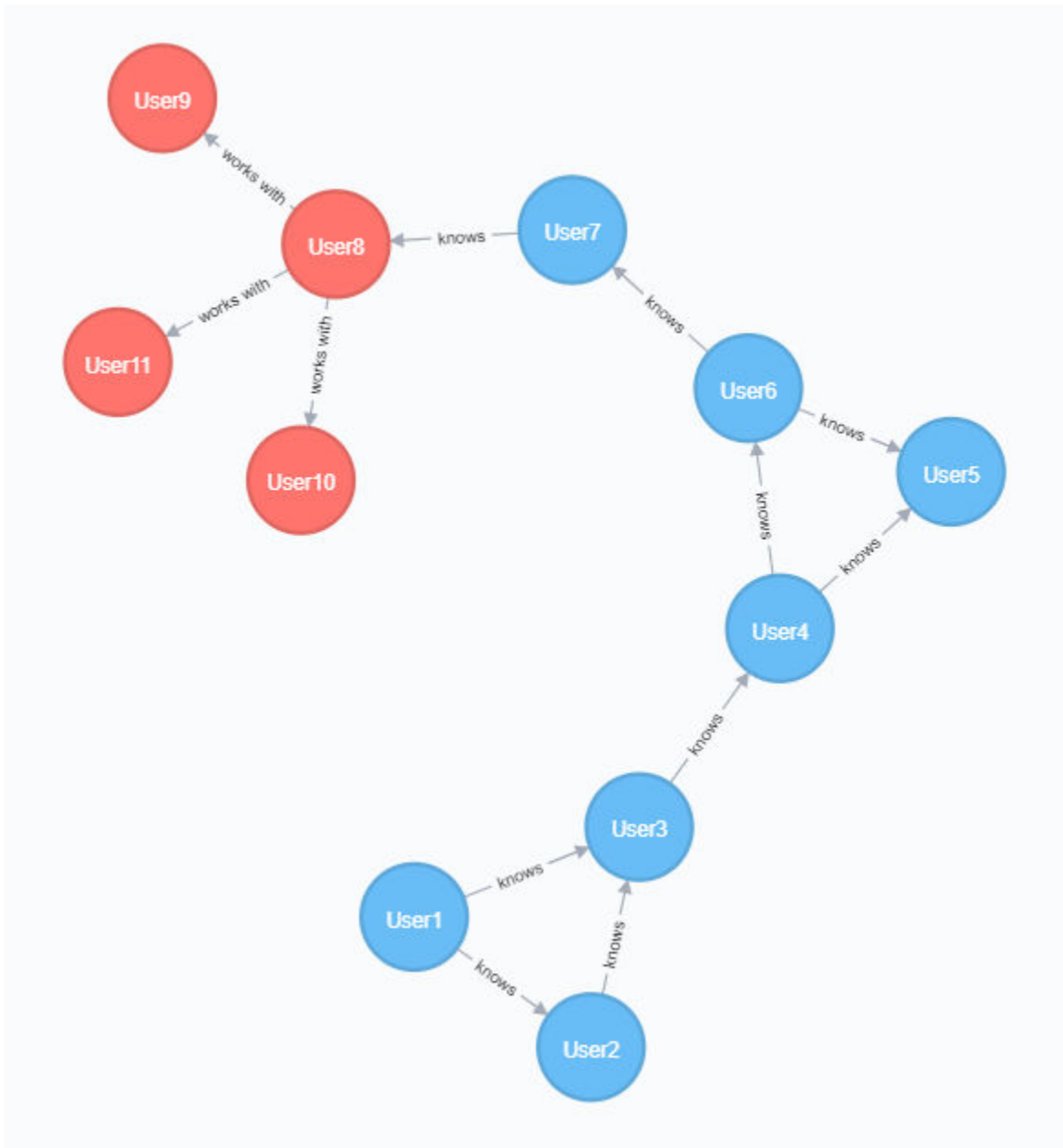
```
nlabel = 'Person';  
user7 = searchNode(neo4jconn,nlabel,'PropertyKey','name', ...
```

```
'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.

See Also

neo4j | storeDigraph

More About

- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12
- “Explore Graph Database Structure” on page 6-2

External Websites

- Neo4j Documentation
- Cypher Query Language

MATLAB Interface to Neo4j Error Messages

Both the MATLAB interface to Neo4j 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 returns error messages in plain text. This table describes how to address common errors you can encounter while working with the MATLAB interface to Neo4j.

Error Message	Probable Causes	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>relationTypes</code> .

Error Message	Probable Causes	Resolution
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.
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 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>

See Also

neo4j | searchGraph

More About

- “Working with MATLAB Interface to Neo4j” on page 6-9
- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12
- “Find Friends of Friends in Social Neighborhood”
- “Explore Graph Database Structure” on page 6-2

External Websites

- [Neo4j Documentation](#)
- [Cypher Query Language](#)

Database Toolbox Interface For MongoDB Topics

- “Import and Analyze Data from MongoDB” on page 7-2
- “Import Filtered Data from MongoDB” on page 7-5
- “Import Large Data from MongoDB” on page 7-8
- “Export MATLAB Data into MongoDB” on page 7-11
- “Import and Export MATLAB Objects Using MongoDB” on page 7-15
- “Database Toolbox Interface for MongoDB Installation” on page 7-19
- “Database Toolbox Interface for MongoDB Error Messages” on page 7-21

Import and Analyze Data from MongoDB

This example shows how to import employee data from a collection in MongoDB into the MATLAB workspace using the Database Toolbox interface for MongoDB. The example then shows how to conduct a simple data analysis based on the imported data.

To run this example, you must first install the Database Toolbox interface for MongoDB. For details, see “Database Toolbox Interface for MongoDB Installation” on page 7-19.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
mongo with properties:
```

```
Database: 'mongotest'  
UserName: ''  
Server: {'dbtb01'}  
Port: 27017  
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection for document retrieval. Retrieve all documents in the collection by using the MongoDB connection. `documents` is a structure array.

```
collection = "employee";  
documents = find(conn,collection);
```

Using all documents, determine the unique department names. `deplist` contains a cell array of character vectors for the department names. The `employee` collection contains seven departments.

```
departments = {documents(:).department};  
deplist = unique(departments)'
```

```
deplist =
```

```
7×1 cell array
```

```
{'Application Engineering'}  
{'Consulting' }  
{'Development' }  
{'Marketing' }  
{'Sales' }  
{'Support' }  
{'Training' }
```

Determine the maximum salary among all employees.

```
salaries = [documents(:).salary];  
max(salaries)
```

```
ans =
```

```
150000
```

Close the MongoDB connection.

`close(conn)`

See Also

`close` | `find` | `isopen` | `max` | `mongo` | `unique`

More About

- “Import Filtered Data from MongoDB” on page 7-5
- “Import Large Data from MongoDB” on page 7-8
- “Export MATLAB Data into MongoDB” on page 7-11
- “Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

- [MongoDB Manual](#)

Import Filtered Data from MongoDB

This example shows how to import flight data from a MongoDB collection into the MATLAB workspace using the Database Toolbox interface for MongoDB. 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.

To run this example, you must first install the Database Toolbox interface for MongoDB. For details, see “Database Toolbox Interface for MongoDB Installation” on page 7-19.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)

conn =

    mongo with properties:

        Database: 'mongotest'
        UserName: ''
        Server: {'dbtb01'}
        Port: 27017
        CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
        TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 fields to retrieve from the collection.

```
collection = "airlinesmall";  
mongoquery = '{"Year":{"$gte:1998,$lt:2000}}';  
fields = ['{"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:
```

```
    x_id  
    Year  
    Month  
    DayOfMonth  
    DayOfWeek  
    DepTime  
    ArrTime
```

Determine the unique years in the data.

```
years = [documents(:).Year];  
unique(years)
```

```
ans =
```

```
    1998    1999
```

Close the MongoDB connection.

`close(conn)`

See Also

`close` | `find` | `isopen` | `mongo` | `unique`

More About

- “Import and Analyze Data from MongoDB” on page 7-2
- “Import Large Data from MongoDB” on page 7-8
- “Export MATLAB Data into MongoDB” on page 7-11
- “Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

- MongoDB Manual

Import Large Data from MongoDB

This example shows how to import a large set of flight data from a MongoDB collection into the MATLAB workspace using the Database Toolbox interface for MongoDB. To avoid out-of-memory issues with the Java heap when retrieving many documents, use a loop to import large data in batches.

To run this example, you must first install the Database Toolbox interface for MongoDB. For details, see “Database Toolbox Interface for MongoDB Installation” on page 7-19.

Connect to MongoDB

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
mongo with properties:
```

```
Database: 'mongotest'  
UserName: ''  
Server: {'dbtb01'}  
Port: 27017  
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `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	285102752	struct	

Close MongoDB Connection

```
close(conn)
```

See Also

[close](#) | [count](#) | [find](#) | [isopen](#) | [mongo](#) | [while](#)

More About

- “Import and Analyze Data from MongoDB” on page 7-2
- “Import Filtered Data from MongoDB” on page 7-5
- “Export MATLAB Data into MongoDB” on page 7-11
- “Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

- [MongoDB Manual](#)

Export MATLAB Data into MongoDB

This example shows how to export table and structure data from the MATLAB workspace into new MongoDB collections using the Database Toolbox interface for MongoDB. The example then shows how to count the number of documents in the collections, remove documents from the collections, and drop the collections.

To run this example, you must first install the Database Toolbox interface for MongoDB. For details, see “Database Toolbox Interface for MongoDB Installation” on page 7-19.

The example uses two data sets: `patients.xls`, which contains patient data, and `tsunamis.xlsx`, which contains tsunami data. You can find files for these data sets in the `toolbox/matlab/demos` folder.

Connect to MongoDB

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
mongo with properties:
```

```
    Database: 'mongotest'  
    Username: ''  
    Server: {'dbtb01'}  
    Port: 27017  
    CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
    TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.

- This database contains 23,485,919 documents.

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 Collections and Export Data into MongoDB

Load the data sets using the `readtable` function. Convert tsunami data to a structure using the `table2struct` function. The MATLAB workspace contains the `patientdata` table and the `tsunamidata` structure.

```
patientdata = readtable('patients.xls');  
data = readtable('tsunamis.xlsx');  
tsunamidata = table2struct(data);
```

Create collections to store the patient and tsunami data using the MongoDB connection.

```
patientcoll = "patients";  
tsunamicoll = "tsunamis";
```

```
createCollection(conn,patientcoll)  
createCollection(conn,tsunamicoll)
```

Export table data into the `patients` collection. `n` contains the number of documents inserted.

```
n = insert(conn,patientcoll,patientdata)
```

```
n =
```

```
    100
```

Export structure data into the `tsunamis` collection. `n` contains the number of documents inserted.

```
n = insert(conn,tsunamicoll,tsunamidata)
```

```
n =
```

```
162
```

Count Documents in Collections

Display the names of all the collections in the `mongotest` database. The new collections `patients` and `tsunamis` appear in the cell array of character vectors.

```
conn.CollectionNames'
```

```
ans =
```

```
9×1 cell array
```

```
 {'airlinesmall' }  
 {'employee'    }  
 {'largedata'   }  
 {'nyctaxi'     }  
 {'patients'    }  
 {'product'     }  
 {'restaurants' }  
 {'tsunamis'    }  
 {'updateCollection'}
```

Count the number of documents in the two new collections.

```
npatients = count(conn,patientcoll)
```

```
ntsunamis = count(conn,tsunamicoll)
```

```
npatients =
```

```
100
```

```
ntsunamis =
```

```
162
```

Remove Documents and Drop Collections

Remove all documents from both collections. `npatients` and `ntsunamis` contain the number of documents removed from each collection.

```
npatients = remove(conn,patientcoll, '{}')
ntsunamis = remove(conn,tsunamicoll, '{}')
```

```
npatients =
```

```
    100
```

```
ntsunamis =
```

```
    162
```

Drop both collections from the mongotest database.

```
dropCollection(conn,patientcoll)
dropCollection(conn,tsunamicoll)
```

Close MongoDB Connection

```
close(conn)
```

See Also

[close](#) | [count](#) | [createCollection](#) | [dropCollection](#) | [insert](#) | [isopen](#) | [mongo](#) | [remove](#)

More About

- “Import and Export MATLAB Objects Using MongoDB” on page 7-15
- “Import and Analyze Data from MongoDB” on page 7-2
- “Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

- [MongoDB Manual](#)

Import and Export MATLAB Objects Using MongoDB

This example shows how to export objects from the MATLAB workspace into MongoDB using the Database Toolbox interface for MongoDB. 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.

To run this example, you must first install the Database Toolbox interface for MongoDB. For details, see “Database Toolbox Interface for MongoDB Installation” on page 7-19.

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 “Class to Represent Structured Data” (MATLAB). To run the code in this example, define the class in the current folder.

The data represents tensile stress or strain measurements. To calculate the elastic modulus of various materials, use this data. 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

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
mongo with properties:
    Database: 'mongotest'
    UserName: ''
    Server: {'dbtb01'}
    Port: 27017
    CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
    TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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,tdcs);  
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.

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 Connection

```
close(conn)
```

See Also

`close` | `createCollection` | `dropCollection` | `find` | `insert` | `isopen` | `mongo` | `remove`

More About

- “Export MATLAB Data into MongoDB” on page 7-11
- “Import and Analyze Data from MongoDB” on page 7-2
- “Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

- [MongoDB Manual](#)

Database Toolbox Interface for MongoDB Installation

To use the Database Toolbox interface for MongoDB, you must first install it. Ensure that the interface supports your MongoDB version and that you have administrator privileges on your machine.

Installation

To install the Database Toolbox interface for MongoDB, 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 MongoDB.
- 3** Install the Database Toolbox interface for MongoDB.

The installer downloads and installs the MongoDB 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 Add-Ons” (MATLAB). For other information, see “Add-Ons” (MATLAB).

Supported MongoDB Versions

The Database Toolbox interface for MongoDB supports these versions of MongoDB:

- MongoDB 2.4
- MongoDB 2.6
- MongoDB 3.0
- MongoDB 3.2
- MongoDB 3.4

See Also

`close` | `find` | `isopen` | `mongo`

More About

- “Import and Analyze Data from MongoDB” on page 7-2
- “Import Filtered Data from MongoDB” on page 7-5
- “Import Large Data from MongoDB” on page 7-8
- “Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

- MongoDB Manual

Database Toolbox Interface for MongoDB Error Messages

This table describes how to address common errors you can encounter while working with the Database Toolbox interface for MongoDB.

Error Category	Error Message	Probable Causes	Resolution
Installation and configuration	Unable to verify MongoDB Java driver installation.	The installation did not install the MongoDB Java driver successfully.	Reinstall the Database Toolbox interface for MongoDB. For details, see “Database Toolbox Interface for MongoDB Installation” on page 7-19.
	Unable to configure MATLAB Java class path.	The installation did not create or modify the Java class path in the MATLAB preferences folder.	Reinstall the Database Toolbox interface for MongoDB using administrator privileges. For details, see “Database Toolbox Interface for MongoDB Installation” on page 7-19.

Error Category	Error Message	Probable Causes	Resolution
	Unable to find MongoDB Java driver file on MATLAB Java class path. Install the Database Toolbox interface for MongoDB again.	After installation, the MATLAB preferences folder is missing the <code>javaclasspath.txt</code> file.	Reinstall the Database Toolbox interface for MongoDB. For details, see “Database Toolbox Interface for MongoDB Installation” on page 7-19.
MongoDB connection	Invalid Mongo connection.	The MongoDB connection is closed.	Use the <code>mongo</code> function to connect to MongoDB.
Input validation	Valid input types for server are string scalars and string array.	In the <code>mongo</code> function, the specified data type for the input argument <code>server</code> is not a string scalar or string array.	Specify the input argument <code>server</code> as a string scalar or string array.
	Each server must have a port.	In the <code>mongo</code> function, the number of servers specified in the <code>mongo</code> function does not match the number of ports specified.	Ensure that you specify a port number for each specified server in the <code>mongo</code> function. For example, if the <code>server</code> input argument is set to <code>{"dbtb01", "dbtb02"}</code> , then the <code>port</code> input argument must have two port numbers, such as <code>[27017, 27018]</code> .

Error Category	Error Message	Probable Causes	Resolution
	Valid input type is cell array of structures.	In the <code>insert</code> function, the <code>documents</code> input argument is specified as a cell array. The cell array contains data types that are not structures.	In the documents input argument of the <code>insert</code> function, specify only structures in the cell array.
MongoDB Java driver	[Mongo Driver Error]: <i>Error message</i> .	The MongoDB Java driver throws an error.	Use the MongoDB Manual to search for the MongoDB error message.
MongoDB CRUD operations	Unable to access MongoDB collection <i>collectionname</i> .	The database does not contain the specified collection.	Access the <code>CollectionNames</code> property of the <code>mongo</code> object to retrieve a list of all collections in the database. Then, specify an existing collection name.
	MongoDB collection <i>collectionname</i> already exists in the database.	The database already contains a collection with the specified name.	Access the <code>CollectionNames</code> property of the <code>mongo</code> object to retrieve a list of all collections in the database. Then, specify a unique name to create a collection.

See Also

`createCollection` | `dropCollection` | `insert` | `mongo` | `remove` | `update`

More About

- “Database Toolbox Interface for MongoDB Installation” on page 7-19
- “Import and Analyze Data from MongoDB” on page 7-2
- “Export MATLAB Data into MongoDB” on page 7-11

External Websites

- [MongoDB Manual](#)

Functions — Alphabetical List

attr

Package: database.odbc

Retrieve attributes of columns in fetched data set

Syntax

```
attributes = attr(curs)
attributes = attr(curs,colnum)
```

Description

`attributes = attr(curs)` retrieves attribute information for all columns in the fetched data set `curs`.

`attributes = attr(curs,colnum)` retrieves attribute information for the column number `colnum` in the fetched data set `curs`.

Examples

Retrieve Attribute Data for a Fetched Data Set

Create a database connection `conn` to an Oracle database using an ODBC connection. This code assumes that you are connecting a data source named `dbname` with user name `username` and password `pwd`. The data source identifies an Oracle database that contains the table `inventoryTable` with these columns: `productNumber`, `Quantity`, `Price`, and `inventoryDate`.

```
conn = database(dbname,username,pwd);
```

Import all the data from the table `inventoryTable`. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
sqlquery = 'SELECT * FROM inventoryTable';
```

```
curs = exec(conn,sqlquery);  
curs = fetch(curs);
```

Retrieve attribute information for all the fetched data using curs.

```
attributes = attr(curs)
```

```
attributes =
```

```
1x4 struct array with fields:
```

```
    fieldName  
    typeName  
    typeValue  
    columnWidth  
    precision  
    scale  
    currency  
    readOnly  
    nullable  
    Message
```

attributes contains a structure array for three columns in the table inventoryTable.

Display the attribute data for the first column in the table inventoryTable.

```
attributes(1)
```

```
ans =
```

```
    fieldName: 'PRODUCTNUMBER'  
    typeName: 'NUMBER'  
    typeValue: 2.00  
columnWidth: 39.00  
    precision: 38.00  
    scale: 0  
    currency: 'true'  
    readOnly: 'false'  
    nullable: 'true'  
    Message: []
```

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)
close(conn)
```

Retrieve Attribute Data for a Specific Column

Create a database connection `conn` to an Oracle database using an ODBC connection. This code assumes that you are connecting a data source named `dbname` with user name `username` and password `pwd`. The data source identifies an Oracle database that contains the table `inventoryTable` with these columns: `productNumber`, `Quantity`, `Price`, and `inventoryDate`.

```
conn = database(dbname,username,pwd);
```

Fetch all the data from the table `inventoryTable`. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');
curs = fetch(curs);
```

Retrieve attribute information for the third column in the table `inventoryTable` using `curs`.

```
attributes = attr(curs,3)
```

```
attributes =
```

```
    fieldName: 'PRICE'
    typeName: 'NUMBER'
    typeValue: 2.00
columnWidth: 39.00
    precision: 126.00
    scale: -127.00
    currency: 'true'
    readOnly: 'false'
    nullable: 'true'
    Message: []
```

`attributes` contains a structure with the attribute data for the third column `PRICE` in the table `inventoryTable`.

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)
close(conn)
```

- “Display Information About Imported Data” on page 5-50

Input Arguments

curs — Database cursor

cursor object

Database cursor, specified as a cursor object created using the `exec` function.

colnum — Column number

numeric scalar

Column number, specified as a numeric scalar to denote the column in the fetched data set `curs` for retrieving attribute information.

Data Types: `double`

Output Arguments

attributes — Attribute data

structure array

Attribute data, returned as a structure array containing attribute information for each column in the fetch data set `curs`. The following attributes are available.

Attribute	Description
<code>fieldName</code>	Name of the column.
<code>typeName</code>	Data type.
<code>typeValue</code>	Numerical representation of the data type.
<code>columnWidth</code>	Size of the field.
<code>precision</code>	Precision value for floating and double data types; an empty value is returned for character vectors or string scalars.

Attribute	Description
scale	Precision value for real and numeric data types; an empty value is returned for character vectors or string scalars.
currency	If this equals <code>true</code> , the data format is currency.
readOnly	If this equals <code>true</code> , the data cannot be overwritten.
nullable	If this equals <code>true</code> , the data can be <code>NULL</code> .
Message	Error message returned by <code>fetch</code> .

See Also

`close` | `cols` | `columnnames` | `columns` | `database` | `dmd` | `fetch` | `get` | `tables` | `width`

Topics

“Display Information About Imported Data” on page 5-50

Introduced before R2006a

catalogs

(To be removed) Get database catalog names

Note The catalogs function has been removed. Use the `DefaultCatalog` and `Catalogs` properties of the connection object instead.

Syntax

```
cn = catalogs(conn)
```

Description

`cn = catalogs(conn)` returns the catalogs for the database connection `conn`.

Examples

Retrieve Catalog Names in the Database

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 user name `username` and password `pwd`.

```
conn = database('dbname', 'username', 'pwd', ...  
              'Vendor', 'MySQL', ...  
              'Server', 'sname');
```

Alternatively, use the native ODBC interface for an ODBC connection. For details, see `database`.

Retrieve the catalog names using `conn`.

```
cn = catalogs(conn)
```

```
cn =  
  
    'toy_store'  
    'mysql'  
    'db'  
    ...
```

`cat` returns a cell array of catalog names in the MySQL database.

Close the connection.

```
close(conn)
```

- “Display Database Metadata” on page 5-38

Input Arguments

conn — Database connection

connection object

Database connection, specified as a `connection` object created with the `database` function.

Output Arguments

cn — Catalog names

cell array

Catalog names, returned as a cell array containing the names of the catalogs in the database. The contents of `cn` that you see depend upon your permission settings in the database.

See Also

`close` | `columns` | `database` | `schemas` | `tables`

Topics

“Display Database Metadata” on page 5-38

Introduced in R2010a

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

First, connect to the Microsoft® SQL Server® database. 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 that contains the imported data from executing the SQL `SELECT` statement.

```
selectquery = 'SELECT * FROM productTable ORDER BY productNumber';  
data = select(conn,selectquery);
```

Display first three rows of data.

```
data(1:3,:)
```

```
ans =
```

```
3x5 table array
```

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 SQLite Connection Object

Create a SQLite connection using the MATLAB® interface to SQLite and the existing database file `tutorial.db`, which resides in the current working 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)
```

Close DatabaseDatastore Object

Using a JDBC driver, create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to database `toy_store`, database server `dbtb04`, and port number `54317`.

```
conn = database('toy_store', '', '', 'Vendor', 'Microsoft SQL Server', ...  
              'Server', 'dbtb04', 'PortNumber', 54317, 'AuthType', 'Windows');
```

Create a `DatabaseDatastore` object using the database connection and SQL query. This SQL query retrieves all data from the table.

```
sqlquery = 'select * from airlinesmall';  
dbds = databaseDatastore(conn, sqlquery);
```

Close the `DatabaseDatastore` object.

```
close(dbds)
```

Close cursor Object

First, connect to the Microsoft® SQL Server® database. 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 `productTable` and sort it by the product number. `curs` is the cursor object that contains the executed SQL query.

```
sqlquery = 'SELECT * FROM productTable ORDER BY productNumber';
curs = exec(conn, sqlquery);
```

Import data from the executed SQL query and display the first three rows.

```
curs = fetch(curs);
curs.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.

```
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)
```

- “Import Data from Databases into MATLAB” on page 5-3
- “Import Large Data Using DatabaseDatastore Object” on page 5-64
- “Export Data to New Record in Database” on page 5-21
- “Display Information About Imported Data” on page 5-50
- “Import Data Using MATLAB® Interface to SQLite” on page 5-68

Input Arguments

object — Database and driver resource utilizer

connection object | sqlite object | DatabaseDatastore object | cursor object

Database and driver resource utilizer, specified as one of these objects.

Object Argument Name	Object Name	Object Description	Object Creation Function
conn	connection	Create a connection between installed database and MATLAB. For details, see “Connecting to Database” on page 2-142.	database
conn	sqlite	Create connection to SQLite database file using the MATLAB interface to SQLite. For details, see “Working with MATLAB Interface to SQLite” on page 2-6.	sqlite
dbds	DatabaseDatastore	Create connection to a type of datastore for working with large data.	databaseDatastore
curs	cursor	Store imported data.	exec

- connection objects, sqlite objects, DatabaseDatastore objects, and cursor objects remain open until you close them using the close function. Always close these objects when you finish using them.
- Close a cursor object before closing the connection used for that cursor object.
- Executing close with a DatabaseDatastore object releases the MATLAB resources associated with connection and cursor objects.

Note The MATLAB session closes open cursor objects, DatabaseDatastore objects, and connections when exiting. However, the database might not free up the cursors and connections.

See Also

database | databaseDatastore | exec | fetch | sqlite

Topics

“Import Data from Databases into MATLAB” on page 5-3

“Import Large Data Using DatabaseDatastore Object” on page 5-64

“Export Data to New Record in Database” on page 5-21

“Display Information About Imported Data” on page 5-50

“Import Data Using MATLAB® Interface to SQLite” on page 5-68

“Configuring Driver and Data Source” on page 2-15

“Connecting to Database” on page 2-142

DatabaseDatastore

“Working with MATLAB Interface to SQLite” on page 2-6

Introduced before R2006a

cols

Package: database.odbcc

Retrieve number of columns in fetched data set

Syntax

```
numcols = cols(curs)
```

Description

`numcols = cols(curs)` returns the number of columns in the fetched data set `curs`.

Examples

Display the Number of Columns in the Data

Create a database connection `conn` using the `dbdemo` data source.

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

Working with the `dbdemo` data source, use `fetch` to import all data into Database Cursor Object `curs`. Store the data in a cell array contained in the cursor object field `curs.Data`.

```
curs = exec(conn, 'SELECT * FROM productTable');  
curs = fetch(curs);
```

View the contents of the `Data` property in the cursor object.

```
curs.Data
```

```
ans =
```

```
    [ 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'
```

Data contains the productTable data.

Display the number of columns in the Data property in the cursor object.

```
numcols = cols(curs)
```

```
numcols =
```

```
5
```

The data in the cursor object contains five columns.

After you finish working with the cursor object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

- “Display Information About Imported Data” on page 5-50

Input Arguments

curs — Database cursor

cursor object

Database cursor, specified as a cursor object created using the exec function.

Output Arguments

numcols — Number of columns

numeric scalar

Number of columns in a data set, returned as a numeric scalar.

See Also

attr | close | columnnames | columnprivileges | columns | database | fetch |
get | rows | width

Topics

“Display Information About Imported Data” on page 5-50

“Connecting to Database Using Native ODBC Interface” on page 3-19

Introduced before R2006a

columnnames

Package: database.odbc

Retrieve names of columns in fetched data set

Syntax

```
columnlist = columnnames(curs)
columnlist = columnnames(curs,returnCellArray)
```

Description

`columnlist = columnnames(curs)` returns the column names of the data selected from a database table in the cursor object `curs`. The `columnnames` function is not supported for a cursor object returned by the `fetchmulti` function.

`columnlist = columnnames(curs,returnCellArray)` returns the column names as a cell array of character vectors when `returnCellArray` is set to `true`.

Examples

Return Column Names from the Selected Data

Create a database connection `conn` using the `dbdemo` data source.

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

Working with the `dbdemo` data source, use `fetch` to import all data into Database Cursor Object `curs`.

```
curs = exec(conn,'SELECT * FROM suppliers');
curs = fetch(curs);
```

Return the column names in the `suppliers` table.

```
columnlist = columnnames(curs)
```

```
columnlist =  
'SupplierNumber','SupplierName','City','Country','FaxNumber'
```

`columnlist` contains one long character vector with the column names in the `suppliers` table in quotes and separated by commas.

After you finish working with the cursor object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

Return Column Names as a Cell Array

Create a database connection `conn` using the `dbdemo` data source.

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

Working with the `dbdemo` data source, use `fetch` to import all data into Database Cursor Object `curs`. Store the data in a cell array contained in the cursor object field `curs.Data`.

```
curs = exec(conn,'SELECT * FROM productTable');  
curs = fetch(curs);
```

Return the column names in the `suppliers` table as a cell array.

```
columnlist = columnnames(curs,true)
```

```
columnlist =  
  
    'SupplierNumber'  
    'SupplierName'  
    'City'  
    'Country'  
    'FaxNumber'
```

`columnlist` contains a cell array of the column names in the `suppliers` table. The cell array has five rows for each column name.

After you finish working with the cursor object, close it.

`close(curs)`

Close the database connection.

`close(conn)`

- “Display Information About Imported Data” on page 5-50

Input Arguments

`curs` — Database cursor

cursor object

Database cursor, specified as a cursor object created using the `exec` function.

`returnCellArray` — Return format

`true` | `false`

Return format, specified as Boolean values `true` or `false`. When set to `true`, `columnnames` returns the column names as a cell array of character vectors. When set to `false`, `columnnames` returns the column names as a long character vector.

Data Types: `logical`

Output Arguments

`columnlist` — Column name list

character vector | cell array

Column name list of columns in the selected data, returned as a character vector or a cell array of character vectors. Without the argument `returnCellArray`, `columnnames` returns the list of column names as a long character vector. The character vector encloses the column names in quotes and separates the column names by commas. If you use the argument `returnCellArray` and set it to `true`, then `columnnames` returns the column names as a cell array.

See Also

`attr` | `close` | `cols` | `columnprivileges` | `columns` | `database` | `fetch` | `get` | `width`

Topics

“Display Information About Imported Data” on page 5-50

Introduced before R2006a

columnprivileges

(To be removed) List database column privileges

Note The `columnprivileges` function will be removed in a future release. Use the `sqlfind` function and access the properties of the `connection` object instead.

Syntax

```
lp = columnprivileges(dbmeta, 'cata', 'sch', 'tab')
lp = columnprivileges(dbmeta, 'cata', 'sch', 'tab', 'l')
```

Description

`lp = columnprivileges(dbmeta, 'cata', 'sch', 'tab')` returns a list of privileges for all columns in the table `tab`, in the schema `sch`, in the catalog `cata` for the database whose database metadata object is `dbmeta`.

`lp = columnprivileges(dbmeta, 'cata', 'sch', 'tab', 'l')` returns a list of privileges for column `l` in the table `tab`, in the schema `sch`, in the catalog `cata` for the database whose database metadata object is `dbmeta`.

Examples

Return a list of privileges for the given database, catalog, schema, table, and column name:

```
lp = columnprivileges(dbmeta, 'msdb', 'geck', 'builds', ...
'build_id')
lp =
    'builds'      'build_id'      {1x4 cell}
```

View the contents of the third column in `lp`:


```
lp{1,3}  
ans =  
    'INSERT'      'REFERENCES'    'SELECT'      'UPDATE'
```

See Also

cols | columnnames | columns | dmd | get | sqlfind

Topics

“Display Database Metadata” on page 5-38

Introduced before R2006a

columns

Package: database.odbc

(To be removed) Return database table column names

Note The `columns` function will be removed in a future release. Use the `sqlfind` function and access the properties of the `connection` object instead.

Syntax

```
columnlist = columns(conn, catalog)
columnlist = columns(conn, catalog, schema)
columnlist = columns(conn, catalog, schema, tablename)
```

```
columnlist = columns(dbmeta, catalog)
columnlist = columns(dbmeta, catalog, schema)
columnlist = columns(dbmeta, catalog, schema, tablename)
```

Description

`columnlist = columns(conn, catalog)` returns a list of all column names in the catalog `catalog` for the database with the database connection `conn`.

`columnlist = columns(conn, catalog, schema)` returns a list of all column names in the schema `schema`.

`columnlist = columns(conn, catalog, schema, tablename)` returns a list of all column names for the table `tablename`.

`columnlist = columns(dbmeta, catalog)` returns a list of all column names in the catalog `catalog` for the database whose database metadata object is `dbmeta`.

`columnlist = columns(dbmeta, catalog, schema)` returns a list of all column names in the schema `schema`.

`columnlist = columns(dbmeta, catalog, schema, tablename)` returns a list of all column names for the table `tablename`.

Examples

Retrieve Column List for Catalog Using Database Connection

Create a database connection `conn` using the native ODBC interface to the Microsoft SQL Server database. 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');
```

Retrieve the column names for each table in a catalog. Here, this code assumes that the database contains the catalog name `toy_store`.

```
catalog = 'toy_store';
```

```
columnlist = columns(conn, catalog)
```

```
columnlist =
```

```
    'salesVolume'           {1x13  cell}
    'suppliers'             {1x5   cell}
    'yearlySales'          {1x3   cell}
    ...
```

`columns` returns a cell array. The first column contains the table names as character vectors. The second column contains the corresponding column name lists as cell arrays.

Display the column names for the `suppliers` table.

```
columnlist{2,2}
```

```
ans =
```

```
    'SupplierNumber'    'SupplierName'    'City'    'Country'    'FaxNumber'
```

Close the database connection.

```
close(conn)
```

Retrieve Column List for Catalog and Schema Using Database Connection

Create a database connection `conn` using the native ODBC interface to the Microsoft SQL Server database. 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');
```

Retrieve the column names for each table in a schema. Here, this code assumes that the database contains the catalog name `toy_store` and the schema name `sch`.

```
catalog = 'toy_store';  
schema = 'sch';
```

```
columnlist = columns(conn,catalog,schema)
```

```
columnlist =
```

```
    'inserttest'           {1x3  cell}  
    'inventoryTable'      {1x4  cell}  
    'largedata'           {1x9  cell}  
    ...
```

`columns` returns a cell array. The first column contains the table names as character vectors. The second column contains the corresponding column name lists as cell arrays.

Display the column names for the `inventoryTable` table.

```
columnlist{2,2}
```

```
ans =
```

```
    'productNumber'    'Quantity'    'Price'    'inventoryDate'
```

Close the database connection.

```
close(conn)
```

Retrieve Column List for Catalog, Schema, and Table Name Using Database Connection

Create a database connection `conn` using the native ODBC interface to the Microsoft SQL Server database. 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');
```

Retrieve the column names in a database table. Here, this code assumes that the database contains the catalog name `toy_store`, the schema name `sch`, and the table name `inventoryTable`.

```
catalog = 'toy_store';
schema = 'sch';
tablename = 'inventoryTable';
```

```
columnlist = columns(conn, catalog, schema, tablename)
```

```
columnlist =
```

```
    'productNumber'    'Quantity'    'Price'    'inventoryDate'
```

`columns` returns a cell array with the column names as character vectors.

Close the database connection.

```
close(conn)
```

Retrieve Column List for Catalog Using Database Metadata Object

Create a database connection `conn`. This code uses database name `dbname`, user name `username`, password `pwd`, database server name `sname`, and port number 123456 to connect to a Microsoft SQL Server database.

```
conn = database('dbname', 'username', 'pwd', ...
    'Vendor', 'Microsoft SQL Server', ...
    'Server', 'sname', ...
    'PortNumber', 123456);
```

Create the database metadata object `dbmeta`.

```
dbmeta = dmd(conn);
```

Retrieve the column names for each table in a catalog. Here, this code assumes that the database contains the catalog name `toy_store`.

```
catalog = 'toy_store';  
columnlist = columns(dbmeta,catalog)  
columnlist =  
    'salesVolume'           {1x13  cell}  
    'suppliers'             {1x5   cell}  
    'yearlySales'          {1x3   cell}  
    ...
```

`columns` returns a cell array. The first column contains the table names as character vectors. The second column contains the corresponding column name lists as cell arrays.

Display the column names for the `suppliers` table.

```
columnlist{2,2}  
ans =  
    'SupplierNumber'    'SupplierName'    'City'    'Country'    'FaxNumber'
```

Close the database connection.

```
close(conn)
```

Retrieve Column List for Catalog and Schema Using Database Metadata Object

Create a database connection `conn`. This code uses database name `dbname`, user name `username`, password `pwd`, database server name `sname`, and port number 123456 to connect to a Microsoft SQL Server database.

```
conn = database('dbname','username','pwd', ...  
    'Vendor','Microsoft SQL Server', ...  
    'Server','sname', ...  
    'PortNumber',123456);
```

Create the database metadata object `dbmeta`.

```
dbmeta = dmd(conn);
```

Retrieve the column names for each table in a schema. Here, this code assumes that the database contains the catalog name `toy_store` and the schema name `sch`.

```
catalog = 'toy_store';
schema = 'sch';

columnlist = columns(dbmeta,catalog,schema)

columnlist =

    'inventoryTable'          {1x4  cell}
    'invoice'                 {1x5  cell}
    'productTable'           {1x5  cell}
    ...
```

`columns` returns a cell array. The first column contains the table names as character vectors. The second column contains the corresponding column name lists as cell arrays.

Display the column names for the `inventoryTable` table.

```
columnlist{1,2}

ans =

    'productNumber'    'Quantity'    'Price'    'inventoryDate'
```

Close the database connection.

```
close(conn)
```

Retrieve Column List for Catalog, Schema, and Table Name Using Database Metadata Object

Create a database connection `conn`. This code uses database name `dbname`, user name `username`, password `pwd`, database server name `sname`, and port number 123456 to connect to a Microsoft SQL Server database.

```
conn = database('dbname','username','pwd', ...
    'Vendor','Microsoft SQL Server', ...
    'Server','sname', ...
    'PortNumber',123456);
```

Create the database metadata object `dbmeta`.

```
dbmeta = dmd(conn);
```

Retrieve the column names in a database table. Here, this code assumes that the database contains the catalog name `toy_store`, the schema name `sch`, and the table name `inventoryTable`.

```
catalog = 'toy_store';  
schema = 'sch';  
tablename = 'inventoryTable';
```

```
columnlist = columns(dbmeta,catalog,schema,tablename)
```

```
columnlist =
```

```
    'productNumber'    'Quantity'    'Price'    'inventoryDate'
```

`columns` returns a cell array with the column names as character vectors.

Close the database connection.

```
close(conn)
```

- “Display Database Metadata” on page 5-38

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with the database function.

dbmeta — Database metadata

database metadata object

Database metadata, specified as a database metadata object created using `dmd`. To use this object, create a database connection using the database function first.

Note Support for the database metadata object will be removed in a future release. Use the `sqlfind` function and access the properties of the connection object instead.

catalog — Database catalog name

character vector | string scalar

Database catalog name, specified as a character vector or string scalar.

Data Types: char | string

schema — Database schema name

character vector | string scalar

Database schema name, specified as a character vector or string scalar.

Data Types: char | string

tablename — Database table name

character vector | string scalar

Database table name, specified as a character vector or string scalar denoting the name of a table in the database.

Example: 'employees'

Data Types: char | string

Output Arguments

columnlist — List of columns

cell array

List of columns, returned as a cell array.

See Also

attr | close | cols | columnnames | columnprivileges | database | dmd | get | sqlfind

Topics

“Display Database Metadata” on page 5-38

“Connecting to Database Using Native ODBC Interface” on page 3-19

Introduced in R2010a

commit

Package: database.odbcc

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`.

See Also

`database` | `datainsert` | `exec` | `get` | `rollback` | `update`

Topics

“Import Data from Databases into MATLAB” on page 5-3

“Export Data to New Record in Database” on page 5-21

“Roll Back Data After Updating Record” on page 5-18

Introduced before R2006a

configureODBCDataSource

Open ODBC Data Source Administrator dialog box

Syntax

```
configureODBCDataSource
```

Description

configureODBCDataSource opens the ODBC Data Source Administrator dialog box on Windows systems.

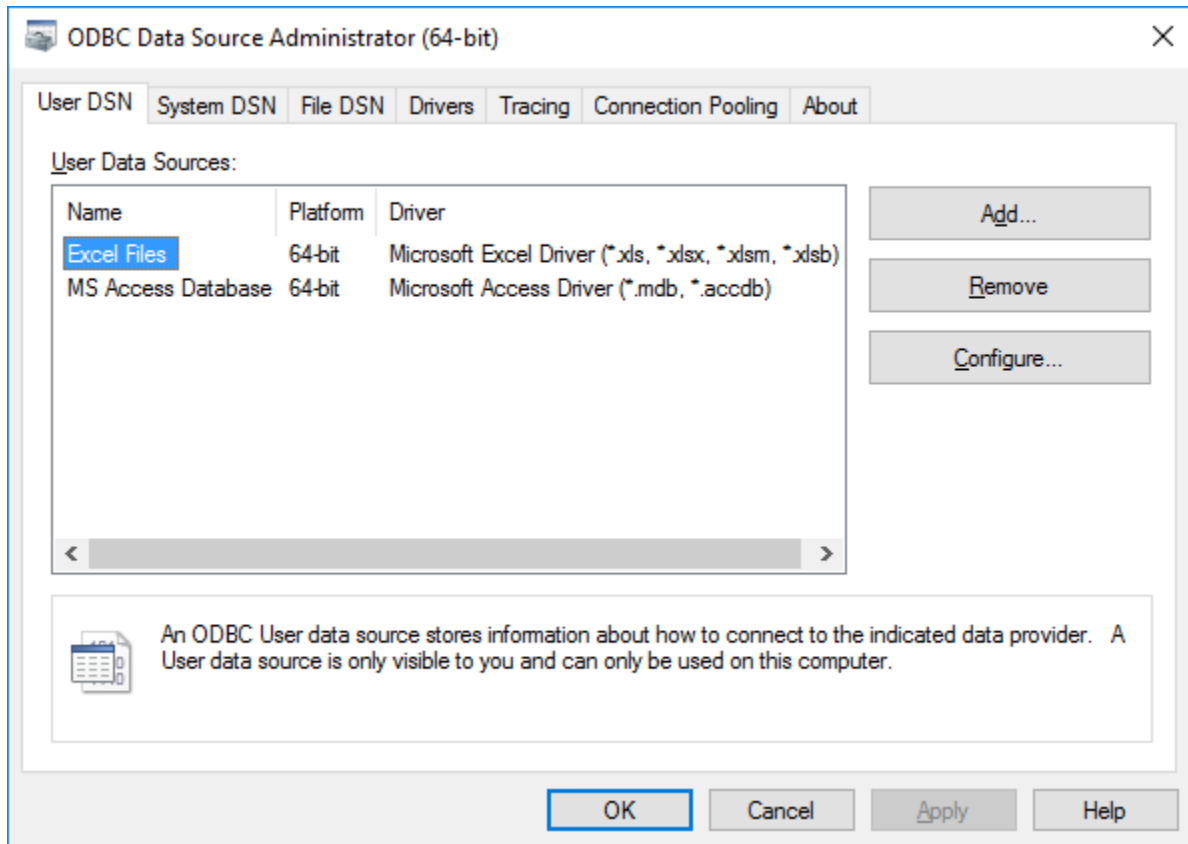
Examples

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.



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 <https://www.mathworks.com/products/matlab/preparing-for-64-bit-windows.html>.

Alternative Functionality

App

You can open the ODBC Data Source Administrator dialog box using the **Database Explorer** app. Select **Configure Data Source > Configure ODBC data source**. For configuring ODBC data sources, see “Configuring Driver and Data Source” on page 2-15.

See Also

Apps

Database Explorer

Functions

database

Topics

“Configuring Driver and Data Source” on page 2-15

Introduced in R2017b

connection

Relational database connection

Description

Create a database connection using either ODBC or JDBC drivers. For information on which connection option is best in your situation, see “Choosing Between ODBC and JDBC Drivers” on page 2-13. Connecting to a database using an ODBC driver uses the native ODBC interface. For details, see “Connecting to Database Using Native ODBC Interface” on page 3-19.

You can use this object to connect to various databases using different drivers that you install and administer. For details, see “Connecting to Database” on page 2-142.

Creation

Create a `connection` object using the `database` function.

Properties

ODBC and JDBC Connection Properties

DataSource — Data source name

' ' (default) | character vector

This property is read-only.

Data source name for ODBC connection or database name for JDBC connection, specified as a character vector. For an ODBC driver, `DataSource` is the name you provide for your data source when you create a data source using the Microsoft ODBC Administrator. For a JDBC driver, `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: 'ODBC Driver Error: [Micro ...']

Data Types: char

Type — Database connection type

'JDBC Connection Object' | 'ODBC Connection Object'

This property is read-only.

Database connection type, specified as one of these values:

- 'JDBC Connection Object' — Database connection is created using a JDBC driver.
- 'ODBC Connection Object' — Database connection is created using an ODBC 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 — Auto-commit transactions

'on' (default) | 'off'

Auto-commit 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.

Data Types: char

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 or JDBC 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 or 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

ODBC and JDBC Connection Functions

close	Close and invalidate database and driver resource utilizer
datainsert	(To be removed) Export MATLAB data into database table

fetch	Import data into MATLAB workspace from database cursor or from execution of SQL statement
rollback	Undo database changes
update	Replace data in database table with MATLAB data
columns	(To be removed) Return database table column names
exec	Execute SQL statement and open cursor
insert	Add MATLAB data to database tables
runsqlscript	Run SQL script on database
commit	Make database changes permanent
fastinsert	(To be removed) Add MATLAB data to database tables
isopen	Determine if database connection or database cursor is open
tables	(To be removed) Return database table names
select	Execute SQL SELECT statement and import data into MATLAB

JDBC Connection Function

runstoredprocedure Call stored procedure with and without input and output arguments

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 =
```

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 from the table.

```
max(data.Quantity)
```

```
ans =
```

```
9000
```

Close the database connection conn.

```
close(conn)
```

Connect to Oracle Using JDBC Driver

First, create a JDBC connection to the Oracle 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 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 the data source name, user name, and password.

```
datasource = 'dsname';  
username = 'username';  
password = 'pwd';
```

```
conn = database(datasource,username,password,'Vendor','Oracle', ...  
               'Server','remotehost','DriverType','thin','PortNumber',1234)
```

```
conn =
```

```
connection with properties:
```

```
DataSource: 'datasource'  
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 from the table.

```
max(data.Quantity)
```

```
ans =  
    9000
```

Close the database connection `conn`.

```
close(conn)
```

Alternative Functionality

A `connection` object is one of the two available database connection types. The other creates a `sqlite` object that connects to a SQLite database file using the MATLAB interface to SQLite without installing or administering a database or driver. For details, see “Working with MATLAB Interface to SQLite” on page 2-6.

See Also

`close` | `exec` | `fetch` | `select`

Topics

“Configuring Driver and Data Source” on page 2-15

“MySQL ODBC for Windows” on page 2-55

“Oracle JDBC for Windows” on page 2-48

“Connecting to Database” on page 2-142

“Import Data from Databases into MATLAB” on page 5-3

“Export Data to New Record in Database” on page 5-21

“Database Connection Error Messages” on page 3-9

Introduced before R2006a

cursor

Database cursor

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. For details, see “Using Scrollable Cursors” on page 5-53.

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 2-145.

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. For details, see “Using Scrollable Cursors” on page 5-53.

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

attr	Retrieve attributes of columns in fetched data set
close	Close and invalidate database and driver resource utilizer
cols	Retrieve number of columns in fetched data set
columnnames	Retrieve names of columns in fetched data set
fetch	Import data into MATLAB workspace from database cursor or from execution of SQL statement
fetchmulti	Import data from multiple result sets
get	(To be removed) Retrieve object properties
isopen	Determine if database connection or database cursor is open
querytimeout	Get time specified for SQL queries to succeed
rows	Return number of rows in fetched data set
set	(To be removed) Set properties for database or cursor object
width	Return field size of column in fetched data set

Examples**Select Data Using Native ODBC Interface**

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.

```
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 =  
  
[]
```

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]
```

With the native ODBC interface, the `Type` property contains the character vector `ODBCCursor Object`. For JDBC connections, this property contains the character vector `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)
```

Select Data Using JDBC Driver

Using the cursor object and JDBC driver, import product data from a MySQL database into MATLAB. Then, determine the highest unit cost among products.

Create a JDBC database connection using the 'Vendor' name-value pair argument in the database function to specify connecting to a MySQL database. Specify a user name and password. Specify the database server name using the 'Server' name-value pair argument.

```
conn = database('dbname','username','pwd','Vendor','MySQL', ...  
              'Server','sname');
```

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:  
  
        Attributes: []  
            Data: 0  
        DatabaseObject: [1x1 database.jdbc.connection]  
            RowLimit: 0  
            SQLQuery: 'SELECT * FROM productTable'  
            Message: []  
            Type: 'Database Cursor Object'  
        ResultSet: [1x1 com.mysql.jdbc.JDBC4ResultSet]  
            Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]  
            Statement: [1x1 com.mysql.jdbc.StatementImpl]  
            Fetch: 0
```

For JDBC connections, the `Type` property contains `Database Cursor Object`. For ODBC connection, this property contains `ODBCCursor 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)
```

See Also

`close` | `database` | `fetch` | `setdbprefs`

Introduced before R2006a

database

Connect to database

Syntax

```
conn = database(datasource,username,password)
conn = database(datasource,username,password,driver,url)
conn = database( ____,Name,Value)
```

Description

`conn = database(datasource,username,password)` creates an ODBC database connection to a data source with a user name and password. The database connection is a connection object.

`conn = database(datasource,username,password,driver,url)` creates a JDBC database connection specified by the JDBC driver name and database connection URL.

`conn = database(____,Name,Value)` includes any of the input argument combinations in the previous syntaxes and adds options that you specify by one or more `Name,Value` pair arguments.

To specify optional database properties for ODBC or JDBC drivers, use the ODBC and JDBC connection options. For example, `conn = database(datasource,username,password,'LoginTimeout',5)`; creates an ODBC connection with a login timeout of 5 seconds.

To create a JDBC connection without using the database connection URL, use the JDBC connection options. For example, `conn = database(datasource,username,password,'Vendor','MySQL','Server','remotehost')`; creates a JDBC connection to a MySQL database.

Note The `database.ODBCConnection` syntax will be removed in a future release. Use the `database` syntax instead.

Examples

Connect to Microsoft® SQL Server® Using ODBC Driver

First, connect to the Microsoft® SQL Server® database. Then, import data from the database into MATLAB®. Perform simple data analysis. Close the database connection.

The code assumes that you are connecting 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, '', '')
```

```
conn =
```

```
connection with properties:
```

```
DataSource: 'MS SQL Server Auth'  
UserName: ''  
Message: ''  
Type: 'ODBC Connection Object'
```

```
Database Properties:
```

```
AutoCommit: 'on'  
ReadOnly: 'off'  
LoginTimeout: 15  
MaxDatabaseConnections: 0
```

```
Catalog and Schema Information:
```

```
DefaultCatalog: 'toy_store'  
Catalogs: {'master', 'msdb', 'tempdb' ... and 1 more}  
Schemas: {'dbo', 'INFORMATION_SCHEMA', 'sys'}
```

```
Database and Driver Information:
```

```
DatabaseProductName: 'Microsoft SQL Server'  
DatabaseProductVersion: '11.00.2100'  
DriverName: 'sqlncli11.dll'
```

```
DriverVersion: '11.00.5058'
```

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);  
data(1:3,:)
```

```
ans =
```

<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'

Determine the highest quantity in the table.

```
max(data.Quantity)
```

```
ans =
```

```
9000
```

Close the database connection.

```
close(conn)
```

Connect to PostgreSQL Using JDBC Driver URL

First, connect to the PostgreSQL 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 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.

```
datasource = 'dbname';  
username = 'username';  
password = 'pwd';  
driver = 'org.postgresql.Driver';  
url = 'jdbc:postgresql://host:port/dbname';  
  
conn = database(datasource,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 Microsoft® SQL Server® Using JDBC Driver with Additional Options

First, connect to the Microsoft® SQL Server® database. Then, import data from the database into MATLAB®. Perform simple data analysis. Close the database connection.

The code assumes that you are connecting to a Microsoft® SQL Server® Version 11.00.2100 database using the Microsoft® SQL Server® JDBC Driver 4.0.2206.100.

Create a database connection to a Microsoft® SQL Server® database with Windows® authentication and a login timeout of 5 seconds. Specify a blank user name and password.

```
datasource = 'toy_store';  
conn = database(datasource, '', '', 'Vendor', 'Microsoft SQL Server', ...  
    'Server', 'dbtb04', 'AuthType', 'Windows', 'PortNumber', 54317, ...  
    'LoginTimeout', 5)
```

```
conn =
```

```
connection with properties:
```

```
DataSource: 'toy_store'  
UserName: ''  
Driver: 'com.microsoft.sqlserver.j ...'  
URL: 'jdbc:sqlserver://dbtb04:5 ...'  
Message: ''  
Type: 'JDBC Connection Object'
```

```
Database Properties:
```

```
AutoCommit: 'on'  
ReadOnly: 'off'  
LoginTimeout: 5  
MaxDatabaseConnections: 0
```

```
Catalog and Schema Information:
```

```
DefaultCatalog: 'toy_store'  
Catalogs: {'master', 'model', 'msdb' ... and 2 more}  
Schemas: {'db_accessadmin', 'db_backupoperator', 'db_datareader'
```

```
Database and Driver Information:
```

```
DatabaseProductName: 'Microsoft SQL Server'  
DatabaseProductVersion: '11.00.2100'
```



```
DriverName: 'Microsoft JDBC Driver 4.0 ...'  
DriverVersion: '4.0.2206.100'
```

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);  
data(1:3,:)
```

```
ans =
```

productNumber	Quantity	Price	inventoryDate
1	1700	15	'2014-09-23'
2	1200	9	'2014-07-08'
3	356	17	'2014-05-14'

Determine the highest quantity in the table.

```
max(data.Quantity)
```

```
ans =
```

```
9000
```

Close the database connection.

`close(conn)`

Input Arguments

datasource — Data source name or database name

character vector | string scalar

Data source name or database name, specified as a character vector or string scalar. Specify a data source for ODBC connection, and the database name for JDBC connection.

- For an ODBC driver, **datasource** is the name you provide for your data source when you create a data source using the Microsoft ODBC Administrator.
- For a JDBC driver, **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.

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`

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 8-70.

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 8-70. 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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Vendor', 'MySQL', 'Server', 'remotehost'` connects to a MySQL database using a JDBC driver on a machine named `remotehost`.

Tip: When creating a JDBC connection using the JDBC connection options, you can omit:

- `'Server'` name-value pair argument when connecting to a database locally
- `'PortNumber'` name-value pair argument when connecting to a database server listening on the default port (except for Oracle connections)

ODBC and JDBC Connection Options

AutoCommit — Auto-commit transactions

`'on'` (default) | `'off'`

Auto-commit transactions, specified as the comma-separated pair consisting of `'AutoCommit'` and 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'

JDBC Connection Options

Vendor — Database vendor

'MySQL' | 'Oracle' | 'Microsoft SQL Server' | 'PostgreSQL'

Database vendor, specified as the comma-separated pair consisting of 'Vendor' and one of these values:

- 'MySQL'
- 'Oracle'
- 'Microsoft SQL Server'

- 'PostgreSQL'

If you are connecting to a database system not listed here, use the `driver` and `url` syntax.

Example: 'Vendor', 'Oracle'

Server — Database server

'localhost' (default) | character vector | string scalar

Database server name or address, specified as the comma-separated pair consisting of 'Server' and a character vector or string scalar.

Example: 'Server', 'remotehost'

Data Types: char | string

PortNumber — Server port number

numeric scalar

Server port number where the server is listening, specified as the comma-separated pair consisting of 'PortNumber' and a numeric scalar.

Example: 'PortNumber', 1234

Data Types: double

AuthType — Authentication type

'Server' (default) | 'Windows'

Authentication type (required only for Microsoft SQL Server), specified as the comma-separated pair consisting of 'AuthType' and one of these values:

- 'Server' — Microsoft SQL Server authentication
- 'Windows' — Windows authentication

Example: 'AuthType', 'Windows'

DriverType — Driver type

'thin' | 'oci'

Driver type (required only for Oracle), specified as the comma-separated pair consisting of 'DriverType' and one of these values:

- 'thin' — Thin driver
- 'oci' — Windows authentication

Example: 'DriverType', 'thin'

Output Arguments

conn — Database connection

connection object

Database connection, returned as a connection object.

Definitions

JDBC Driver Name and Database Connection URL

The JDBC driver name and database connection URL take different forms for different databases.

Database	JDBC Driver Name and Database URL Example Syntax
IBM® Informix®	JDBC driver: com.informix.jdbc.IfxDriver Database URL: jdbc:informix-sqli://161.144.202.206:3000:INFORMIXSERVER=stars
Microsoft SQL Server 2005	JDBC driver: com.microsoft.sqlserver.jdbc.SQLServerDriver Database URL: jdbc:sqlserver://localhost:port;database=databasename

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>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>

Database	JDBC Driver Name and Database URL Example Syntax
PostgreSQL with SSL Connection	<p>JDBC driver: org.postgresql.Driver</p> <p>Database URL: jdbc:postgresql:servername:dbname:ssl=true&sslfactory=org.postgresql.ssl.NonValidatingFactory&</p> <p><i>The trailing & is required.</i></p>

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.

See Also

Functions

close | datainsert | exec | fetch | isopen | select | update

Apps

Database Explorer

Topics

“Setup Requirements for Database Connection” on page 2-12

“Connection Options” on page 2-9

“Configuring Driver and Data Source” on page 2-15

“Microsoft SQL Server ODBC for Windows” on page 2-24

“Microsoft SQL Server JDBC for Windows” on page 2-32

“PostgreSQL JDBC for Windows” on page 2-72

“Connecting to Database” on page 2-142

“Connecting to Database Using Native ODBC Interface” on page 3-19

“Import Data from Databases into MATLAB” on page 5-3

“Export Data to New Record in Database” on page 5-21

“Display Database Metadata” on page 5-38

“Database Connection Error Messages” on page 3-9

“Java Class Path” (MATLAB)

Introduced before R2006a

datastore

(Not recommended) Create datastore to access collection of data in a database

Note datastore is not recommended. Use `databaseDatastore` instead.

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” (MATLAB).

Syntax

```
dbds = datastore(conn,sqlquery)
```

Description

`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)
```

- “Import Large Data Using DatabaseDatastore Object” on page 5-64

Input Arguments

conn — Database connection

connection object

Database connection, specified as a `connection` object created with 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.

See Also

[close](#) | [database](#) | [databaseDatastore](#) | [datastore](#) | [preview](#) | [read](#)

Topics

“Import Large Data Using DatabaseDatastore Object” on page 5-64

DatabaseDatastore

“Getting Started with Datastore” (MATLAB)

Introduced in R2014b

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).

Note If there is no more data to read from the query, `hasdata` returns logical 0.

Examples

Determine If DatabaseDatastore Object Contains More Data

Using a JDBC driver, create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to database `toy_store`, database server `dbtb04`, and port number 54317.

```
conn = database('toy_store','','','Vendor','Microsoft SQL Server', ...  
              'Server','dbtb04','PortNumber',54317,'AuthType','Windows');
```

Create a DatabaseDatastore object using the database connection and SQL query. This SQL query reads the first 30 rows of data from the table.

```
sqlquery = 'select top 30 * from airlinesmall';  
dbds = databaseDatastore(conn,sqlquery);
```

Read the first 10 rows.

```
dbds.ReadSize = 10;
read(dbds)
```

ans =

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRS
1990	9	4	2	1228	1230	1350	134
1990	9	12	3	1125	1125	1231	123
1990	9	23	7	1721	1719	2201	220
1990	9	27	4	645	645	802	80
1990	9	3	1	710	711	837	84
1990	9	20	4	1338	1335	1853	190
1990	9	22	6	900	900	1241	122
1990	9	3	1	925	755	1258	114
1990	9	29	6	1434	1435	1615	163
1990	9	2	7	NaN	1805	NaN	190

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 =

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRS
1990	9	4	2	1228	1230	1350	13
1990	9	12	3	1125	1125	1231	12
1990	9	23	7	1721	1719	2201	22
1990	9	27	4	645	645	802	8
1990	9	3	1	710	711	837	8
1990	9	20	4	1338	1335	1853	19
1990	9	22	6	900	900	1241	12
1990	9	3	1	925	755	1258	11
1990	9	29	6	1434	1435	1615	16
1990	9	2	7	NaN	1805	NaN	19

ans =

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRS
1990	9	23	7	1721	1719	2201	22
1990	9	27	4	645	645	802	8
1990	9	3	1	710	711	837	8
1990	9	20	4	1338	1335	1853	19
1990	9	22	6	900	900	1241	12
1990	9	3	1	925	755	1258	11
1990	9	29	6	1434	1435	1615	16
1990	9	2	7	NaN	1805	NaN	19
1990	9	11	2	908	910	1613	15
1990	9	22	6	1801	1750	2005	19

When no more data remains in `dbds`, `hasdata` returns logical `0` and the `while` loop stops.

Close the `DatabaseDatastore` object and database connection.

```
close(dbds)
```

- “Import Large Data Using DatabaseDatastore Object” on page 5-64
- “Analyze Large Data in Database Using Tall Arrays”
- “Analyze Large Data in Database Using MapReduce”

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.

See Also

close | database | databaseDatastore | read

Topics

“Import Large Data Using DatabaseDatastore Object” on page 5-64

“Analyze Large Data in Database Using Tall Arrays”

“Analyze Large Data in Database Using MapReduce”

DatabaseDatastore

Introduced in R2014b

isopen

Package: database.odbc

Determine if database connection or database cursor is open

Syntax

```
i = isopen(conn)
i = isopen(curs)
```

Description

`i = isopen(conn)` returns 1 if the database connection is open and 0 if the database connection is closed or invalid.

`i = isopen(curs)` returns the same values using the database cursor object.

Examples

Determine If Database Connection Is Open

First, connect to the Microsoft® SQL Server® database. 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` that denotes the database connection is open.

```
i = isopen(conn)
```

```
i =
```

```
    1
```

Select all data from `productTable` and sort it by the product number. `data` is a table that contains the imported data from executing the SQL `SELECT` statement.

```
selectquery = 'SELECT * FROM productTable ORDER BY productNumber';
data = select(conn,selectquery);
```

Display 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)
```

Determine if the database connection is closed. The `isopen` function returns the numeric scalar `0` that denotes the database connection is closed. If the database connection is invalid, the `isopen` function returns the same result.

```
i = isopen(conn)
```

```
i =  
    0
```

Determine If Database Cursor Is Open

First, connect to the Microsoft® SQL Server® database. Verify the database cursor. 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 =
```

```
[]
```

Create a cursor object by executing an SQL query in the database. Select all data from the table `productTable` using the database connection.

```
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 that denotes the database cursor is open.

```
i = isopen(curs)
```

```
i =
    1
```

Import data from the executed SQL query. Display the first three rows of imported data.

```
curs = fetch(curs);
curs.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.

```
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` that denotes 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)
```

- “Import Data from Databases into MATLAB” on page 5-3
- “Export Data to New Record in Database” on page 5-21

Input Arguments

conn — Database connection

connection object

Database connection, specified as a `connection` object created with the `database` function.

curs — Database cursor

cursor object

Database cursor, specified as a `cursor` object created using the `exec` function.

See Also

close | database | exec | isreadonly | ping

Topics

“Import Data from Databases into MATLAB” on page 5-3

“Export Data to New Record in Database” on page 5-21

“Connecting to Database” on page 2-142

“Connecting to Database Using Native ODBC Interface” on page 3-19

Introduced in R2015b

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

Using a JDBC driver, create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to database `toy_store`, database server `dbtb04`, and port number 54317.

```
conn = database('toy_store', '', '', 'Vendor', 'Microsoft SQL Server', ...  
              'Server', 'dbtb04', 'PortNumber', 54317, 'AuthType', 'Windows');
```

Create a `DatabaseDatastore` object using the database connection and SQL query. This SQL query reads all data from the 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 =
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1990	9	22	6	1801	1750	2005	1954
1990	9	11	2	908	910	1613	1553
1990	9	2	7	NaN	1805	NaN	1954
1990	9	29	6	1434	1435	1615	1634
1990	9	3	1	925	755	1258	1144
1990	9	22	6	900	900	1241	1222
1990	9	20	4	1338	1335	1853	1904
1990	9	3	1	710	711	837	844

Close the DatabaseDatastore object and database connection.

```
close(dbds)
```

- “Import Large Data Using DatabaseDatastore Object” on page 5-64
- “Analyze Large Data in Database Using Tall Arrays”
- “Analyze Large Data in Database Using MapReduce”

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.

See Also

`close` | `database` | `databaseDatastore` | `read`

Topics

“Import Large Data Using DatabaseDatastore Object” on page 5-64

“Analyze Large Data in Database Using Tall Arrays”

“Analyze Large Data in Database Using MapReduce”

`DatabaseDatastore`

Introduced in R2014b

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

Using a JDBC driver, create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The

code assumes that you are connecting to database `toy_store`, database server `dbtb04`, and port number `54317`.

```
conn = database('toy_store', '', '', 'Vendor', 'Microsoft SQL Server', ...  
              'Server', 'dbtb04', 'PortNumber', 54317, 'AuthType', 'Windows');
```

Create a `DatabaseDatastore` object using the database connection and SQL query. This SQL query retrieves all data from the 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 =
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	28	3	1140	1140	1212	1212
1987	10	9	5	1155	1155	1250	1300
1987	10	22	4	715	715	807	807
1987	10	16	5	1553	1555	1641	1641
1987	10	30	5	1821	1815	1956	1956
1987	10	12	1	1300	1300	1529	1529
1987	10	7	3	810	810	904	904
1987	10	19	1	733	735	827	830
1987	10	15	4	828	830	916	920
1987	10	4	7	1750	1735	1837	1837

`data` contains the query results.

Close the `DatabaseDatastore` object and database connection.

```
close(dbds)
```

Read Data and Database Information

Using a JDBC driver, create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to database `toy_store`, database server `dbtb04`, and port number `54317`.

```
conn = database('toy_store','','','Vendor','Microsoft SQL Server', ...
               'Server','dbtb04','PortNumber',54317,'AuthType','Windows');
```

Create a `DatabaseDatastore` object using the database connection and SQL query. This SQL query retrieves all data from the 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 =
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	28	3	1140	1140	1212	1212
1987	10	9	5	1155	1155	1250	1300
1987	10	22	4	715	715	807	807
1987	10	16	5	1553	1555	1641	1641
1987	10	30	5	1821	1815	1956	1956
1987	10	12	1	1300	1300	1529	1529
1987	10	7	3	810	810	904	904
1987	10	19	1	733	735	827	833
1987	10	15	4	828	830	916	922
1987	10	4	7	1750	1735	1837	1837

```
info =
```

```
struct with fields:
```

```
datasource: 'toy_store'  
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 database connection.

```
close(dbds)
```

- “Import Large Data Using DatabaseDatastore Object” on page 5-64
- “Analyze Large Data in Database Using Tall Arrays”
- “Analyze Large Data in Database Using MapReduce”

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
datasource	Data source name for ODBC drivers or a database name for JDBC drivers
offset	Current cursor position in the returned data set

See Also

[close](#) | [database](#) | [DatabaseDatastore](#) | [hasdata](#)

Topics

[“Import Large Data Using DatabaseDatastore Object”](#) on page 5-64

[“Analyze Large Data in Database Using Tall Arrays”](#)

[“Analyze Large Data in Database Using MapReduce”](#)

[DatabaseDatastore](#)

Introduced in R2014b

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.

Note `readall` returns data as a table only. `readall` ignores database preference settings for data return formatting.

Examples

Read All Data in DatabaseDatastore Object

Using a JDBC driver, create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to database `toy_store`, database server `dbtb04`, and port number 54317.

```
conn = database('toy_store', '', '', 'Vendor', 'Microsoft SQL Server', ...  
              'Server', 'dbtb04', 'PortNumber', 54317, 'AuthType', 'Windows');
```

Create a DatabaseDatastore object using the database connection and SQL query. This SQL query reads all data from the 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 =
```

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
1987	10	28	3	1140	1140	1212	1212
1987	10	9	5	1155	1155	1250	1300
1987	10	22	4	715	715	807	807

Close the DatabaseDatastore object and database connection.

```
close(dbds)
```

- “Import Large Data Using DatabaseDatastore Object” on page 5-64
- “Analyze Large Data in Database Using Tall Arrays”
- “Analyze Large Data in Database Using MapReduce”

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.

See Also

[close](#) | [database](#) | [databaseDatastore](#) | [preview](#) | [read](#)

Topics

[“Import Large Data Using DatabaseDatastore Object”](#) on page 5-64

[“Analyze Large Data in Database Using Tall Arrays”](#)

[“Analyze Large Data in Database Using MapReduce”](#)

[DatabaseDatastore](#)

Introduced in R2014b

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 re-reading from the same DatabaseDatastore.

Examples

Reset DatabaseDatastore Object to Initial State

Using a JDBC driver, create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to database `toy_store`, database server `dbtb04`, and port number 54317.

```
conn = database('toy_store',' ',' ','Vendor','Microsoft SQL Server', ...  
              'Server','dbtb04','PortNumber',54317,'AuthType','Windows');
```

Create a DatabaseDatastore object using the database connection and SQL query. This SQL query retrieves all data from the table. Specify reading a maximum of 10 records from the executed SQL query when using the `read` function.

```
sqlquery = 'select * from airlinesmall';
```

```
dbds = databaseDatastore(conn,sqlquery,'ReadSize',10);
```

Read data from the start of the data set.

`read(dbds)`

ans =

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRS
1987	10	28	3	1140	1140	1212	12
1987	10	9	5	1155	1155	1250	13
1987	10	22	4	715	715	807	8
1987	10	16	5	1553	1555	1641	16
1987	10	30	5	1821	1815	1956	19
1987	10	12	1	1300	1300	1529	15
1987	10	7	3	810	810	904	9
1987	10	19	1	733	735	827	8
1987	10	15	4	828	830	916	9
1987	10	4	7	1750	1735	1837	18

`read` returns the first 10 records in the data set.

Reset the `DatabaseDatastore` object to the state where no data has been read from it. Resetting allows rereading from the same `DatabaseDatastore` object.

`reset(dbds)`

Read data from the start of the data set.

`read(dbds)`

ans =

Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRS
1987	10	28	3	1140	1140	1212	12
1987	10	9	5	1155	1155	1250	13
1987	10	22	4	715	715	807	8
1987	10	16	5	1553	1555	1641	16
1987	10	30	5	1821	1815	1956	19
1987	10	12	1	1300	1300	1529	15
1987	10	7	3	810	810	904	9
1987	10	19	1	733	735	827	8

1987	10	15	4	828	830	916	92
1987	10	4	7	1750	1735	1837	1837

`read` again returns the first 10 records in the data set.

Close the `DatabaseDatastore` object and the database connection.

```
close(dbds)
```

- “Import Large Data Using DatabaseDatastore Object” on page 5-64
- “Analyze Large Data in Database Using Tall Arrays”
- “Analyze Large Data in Database Using MapReduce”

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.

See Also

`close` | `database` | `databaseDatastore` | `exec` | `read`

Topics

“Import Large Data Using DatabaseDatastore Object” on page 5-64

“Analyze Large Data in Database Using Tall Arrays”

“Analyze Large Data in Database Using MapReduce”

`DatabaseDatastore`

Introduced in R2014b

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 the results from executing an SQL query in a relational database. For details about the other datastores, see “Getting Started with Datastore” (MATLAB).

With a `DatabaseDatastore` object, you can preview and read records or chunks in a data set and reset the `DatabaseDatastore` to the initial state. Additionally, 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 `exec` and `fetch` functions on the data set. Advantages are:

- Working with databases containing large amounts of data.
- Analyzing 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” (MATLAB).
- Writing MapReduce algorithms that define the chunking and reduction of large amounts of data using the `mapreduce` function. For details, see “Getting Started with MapReduce” (MATLAB). For an example, see “Analyze Large Data in Database Using MapReduce”. For more MapReduce examples, see `Building Effective Algorithms with MapReduce` (MATLAB).

Creation

Syntax

```
dbds = databaseDatastore(conn,sqlquery)
dbds = databaseDatastore(conn,sqlquery,'ReadSize',rowcount)
```

Description

`dbds = databaseDatastore(conn,sqlquery)` creates a `DatabaseDatastore` object `dbds` using the database connection `conn`. This datastore contains query results from the executed SQL query `sqlquery`.

`dbds = databaseDatastore(conn,sqlquery,'ReadSize',rowcount)` specifies the number of records `rowcount` to return when reading data from the database.

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with 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`

rowcount — Record count

numeric scalar

Record count, specified as a nonnegative numeric scalar to denote the maximum number of records to retrieve from the `DatabaseDatastore` object `dbds`.

Data Types: `double`

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 run MapReduce algorithms, set the global execution environment to be the local MATLAB session using `mapreducer`. Enter this code:

```
mapreducer(0)
```

For details about controlling parallel resources, see “Run mapreduce on a Parallel Pool” (Parallel Computing Toolbox).

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 to denote the SQL query to execute in the database.

Data Types: `char`

VariableNames — Column names of retrieved data table

cell array of character vectors

This property is read-only.

Column names of the retrieved data table, specified as a cell array of one or more character vectors.

Data Types: `char`

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

<code>hasdata</code>	Determine if data in <code>DatabaseDatastore</code> is available to read
<code>preview</code>	Return subset of data from <code>DatabaseDatastore</code>
<code>read</code>	Read data in <code>DatabaseDatastore</code>
<code>readall</code>	Read all data in <code>DatabaseDatastore</code>
<code>reset</code>	Reset <code>DatabaseDatastore</code> to initial state
<code>close</code>	Close and invalidate database and driver resource utilizer

Examples

Create DatabaseDatastore Object

Using a JDBC driver, create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to database `toy_store`, database server `dbtb04`, and port number `54317`.

```
conn = database('toy_store','','','Vendor','Microsoft SQL Server', ...
    'Server','dbtb04','PortNumber',54317,'AuthType','Windows');
```

Create a `DatabaseDatastore` object using a database connection and 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.jdbc.connection]
  Query: 'select * from airlinesmall'
VariableNames: {1x29 cell}
  ReadSize: 10000
```

dbds contains 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: 'toy_store'
  UserName: ''
  Driver: 'com.microsoft.sqlserver.j ...'
  URL: 'jdbc:sqlserver://dbtb04:5 ...'
  Message: ''
  Type: 'JDBC Connection Object'
```

```
Database Properties:
```

```
AutoCommit: 'on'
  ReadOnly: 'off'
  LoginTimeout: 0
  MaxDatabaseConnections: 0
```

```
Catalog and Schema Information:
```

```
DefaultCatalog: 'toy_store'
  Catalogs: {'master', 'model', 'msdb' ... and 2 more}
  Schemas: {'db_accessadmin', 'db_backupoperator', 'db_datareader'
```

```
Database and Driver Information:
```

```

DatabaseProductName: 'Microsoft SQL Server'
DatabaseProductVersion: '11.00.2100'
DriverName: 'Microsoft JDBC Driver 4.0 ...'
DriverVersion: '4.0.2206.100'

```

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	CRS
1990	9	11	2	1810	1812	1939	193
1990	10	27	6	1353	1355	1634	164
1990	10	23	2	1057	1055	1205	115
1990	10	8	1	1515	1440	1609	153
1990	10	19	5	1130	1120	1203	115
1990	10	12	5	1755	1733	1858	182
2001	11	22	4	1345	1355	1530	154
2001	11	26	1	2105	2110	2209	223

Close the DatabaseDatastore object and the database connection.

```
close(dbds)
```

Create DatabaseDatastore Object with Specific Record Count

Using a JDBC driver, create a database connection to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to database `toy_store`, database server `dbtb04`, and port number `54317`.

```
conn = database('toy_store', '', '', 'Vendor', 'Microsoft SQL Server', ...  
              'Server', 'dbtb04', 'PortNumber', 54317, 'AuthType', 'Windows');
```

Create a `DatabaseDatastore` object using a database connection and 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 when using the `read` function. `databaseDatastore` executes the SQL query.

```
sqlquery = 'select * from airlinesmall';
```

```
dbds = databaseDatastore(conn, sqlquery, 'ReadSize', 1000)
```

```
dbds =
```

```
DatabaseDatastore with properties:
```

```
    Connection: [1x1 database.jdbc.connection]  
      Query: 'select * from airlinesmall'  
VariableNames: {1x29 cell}  
      ReadSize: 1000
```

`dbds` contains 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: 'toy_store'  
UserName: ''  
Driver: 'com.microsoft.sqlserver.j ...'  
URL: 'jdbc:sqlserver://dbtb04:5 ...'  
Message: ''
```

```

Type: 'JDBC Connection Object'
Database Properties:
    AutoCommit: 'on'
    ReadOnly: 'off'
    LoginTimeout: 0
    MaxDatabaseConnections: 0

Catalog and Schema Information:
    DefaultCatalog: 'toy_store'
    Catalogs: {'master', 'model', 'msdb' ... and 2 more}
    Schemas: {'db_accessadmin', 'db_backupoperator', 'db_datareader'

Database and Driver Information:
    DatabaseProductName: 'Microsoft SQL Server'
    DatabaseProductVersion: '11.00.2100'
    DriverName: 'Microsoft JDBC Driver 4.0 ...'
    DriverVersion: '4.0.2206.100'

```

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	CRS
1990	9	11	2	1810	1812	1939	193
1990	10	27	6	1353	1355	1634	164
1990	10	23	2	1057	1055	1205	115
1990	10	8	1	1515	1440	1609	153
1990	10	19	5	1130	1120	1203	115
1990	10	12	5	1755	1733	1858	182
2001	11	22	4	1345	1355	1530	154

2001

11

26

1

2105

2110

2209

22

Close the `DatabaseDatastore` object and the database connection.

```
close(dbds)
```

- “Import Large Data Using DatabaseDatastore Object” on page 5-64
- “Analyze Large Data in Database Using Tall Arrays”
- “Analyze Large Data in Database Using MapReduce”

See Also

database | exec

Topics

“Import Large Data Using DatabaseDatastore Object” on page 5-64

“Analyze Large Data in Database Using Tall Arrays”

“Analyze Large Data in Database Using MapReduce”

“Getting Started with Datastore” (MATLAB)

“Getting Started with MapReduce” (MATLAB)

Introduced in R2014b

datainsert

Package: database.odbc

(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.

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 = {'productName', '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)
```

- “Export Data to New Record in Database” on page 5-21
- “Export Multiple Records from MATLAB Workspace” on page 5-27
- “Export Data Using Bulk Insert” on page 5-32
- “Replace Existing Data in Database” on page 5-25
- “Roll Back Data After Updating Record” on page 5-18

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with the `database` function.

tablename — Database table name

character vector | string scalar

Database table name, specified as a character vector or string scalar denoting the name of a table in the database.

Example: `'employees'`

Data Types: `char` | `string`

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 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`.

For other differences among these functions, see “Inserting Data Using Command Line” on page 2-148.

See Also

`close` | `database` | `fastinsert` | `insert` | `select` | `sqlwrite`

Topics

“Export Data to New Record in Database” on page 5-21

“Export Multiple Records from MATLAB Workspace” on page 5-27

“Export Data Using Bulk Insert” on page 5-32

“Replace Existing Data in Database” on page 5-25

“Roll Back Data After Updating Record” on page 5-18

“Inserting Data Using Command Line” on page 2-148

“Data Type Support” on page 1-3

External Websites

SQL Tutorial

Introduced in R2011a

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.

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 selection using SQL query criteria.
- Preview selected data.
- 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

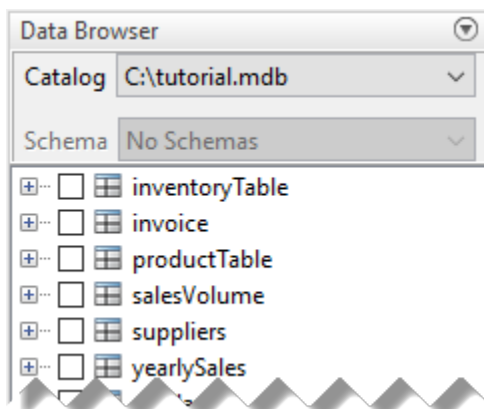
Connect to a Microsoft Access database using the Database Explorer app. Then, select columns from a single table and preview the data. The Database Explorer app previews query results by default.

Set up the data source for the `tutorial.mdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-18.

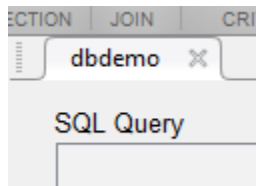
In the **Data Source** section of the **Database Explorer** tab, click **New Query**. The Connect to a Data Source dialog box opens. Select `dbdemo` from the **Data Source** list. Leave the user name and password blank, and click **Connect**.

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 **Data Browser** pane displays the available tables in the database.



The data source tab, which is named **dbdemo**, appears to the right of the **Data 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 **Data 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 **Data 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 **Data Browser** pane.

Select the table name **inventoryTable**.

To change the data you see, select or clear check boxes in the **Data Browser** pane. The SQL query updates in the **SQL Query** pane. The data updates 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 Database Explorer application window. The toolbar includes buttons for 'Configure Data Source', 'New Query', 'Clear Query', 'Manual', 'Exclude Duplicates', 'Join', 'Where', 'Order By', 'Preview Size' (set to 20), 'Automatic Preview', 'Preview Query', and 'Import Data'. The 'Criteria' section is active, and the 'Order By' tab is selected in the toolbar. The SQL Query pane contains the following query:

```
SELECT *
FROM inventoryTable
```

The Data Preview pane shows the results of the query, sorted by the 'Price' column in ascending order. The preview includes 13 rows of data.

	productNumber	Quantity	Price	inventoryDate
1	1	1700	14.5000	2014-09-23 0...
2	2	1200	9	2014-07-08 2...
3	3	356	17	2014-05-14 0...
4	4	2580	21	2013-06-08 1...
5	5	9000	3	2012-09-14 1...
6	6	4540	8	2013-12-25 1...
7	7	6034	16	2014-08-06 0...
8	8	8350	5	2011-06-18 1...
9	9	2339	13	2011-02-09 1...

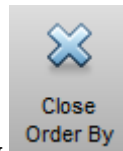
You can sort the rows of data by a specific column. In the **Criteria** section, click **Order By**. The **Order By** tab is displayed in the toolbar.

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 'ORDER BY' dialog box in SQL Server Enterprise Manager. The 'Column' list contains 'Price' and 'Price DESC'. The 'Sort' dropdown is set to 'Descending'. The 'Data Preview' section shows a table with 9 rows of data.

	productNumber	Quantity	Price	inventoryDate
1	10	723	24	2012-03-14 1...
2	4	2580	21	2013-06-08 1...
3	3	356	17	2014-05-14 0...
4	7	6034	16	2014-08-06 0...
5	13	1700	14.5000	2009-05-24 1...
6	1	1700	14.5000	2014-09-23 0...
7	9	2339	13	2011-02-09 1...
8	2	1200	9	2014-07-08 2...
9	6	4540	8	2013-12-25 1...

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.

Close the database connection by closing the data source tab.

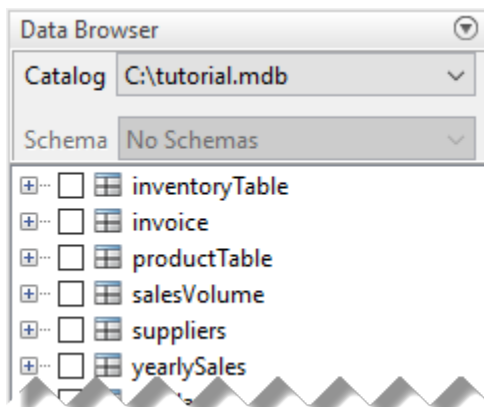
Join Multiple Tables and Import Query Results

Connect to a Microsoft Access database using the Database Explorer app. Then, select columns from multiple tables. 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.

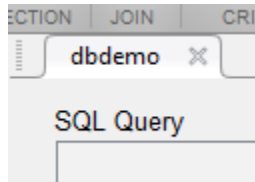
Set up the data source for the `tutorial.mdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-18.

In the **Data Source** section of the **Database Explorer** tab, click **New Query**. The Connect to a Data Source dialog box opens. Select `dbdemo` from the **Data Source** list. Leave the user name and password blank, and click **Connect**.

The Database Explorer app creates a connection to the Microsoft Access database. The **Data Browser** pane displays the available tables in the database.



The data source tab, which is named **dbdemo**, appears to the right of the **Data Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.



In the **Data 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 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.

The screenshot shows the Microsoft Access Database Explorer interface. The title bar reads "Database Explorer - dbdemo". The main toolbar includes buttons for "Configure Data Source", "New Query", "Clear Query", "Manual", "Exclude Duplicates", "Join", "Where", "Order By", "Preview Size" (set to 10), "Automatic Preview", "Preview Query", and "Import Data". Below the toolbar is the "Data Browser" pane on the left, showing a tree view of the database structure. The "Catalog" is set to "C:\tutorial.mdb" and the "Schema" is "No Schemas". The "inventoryTable" is selected and checked. The main area displays the "SQL Query" editor with the following query:

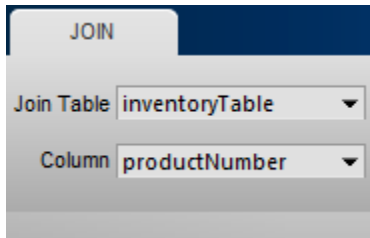
```
SELECT *
FROM inventoryTable
```

Below the query editor is the "Data Preview (First 10 Rows)" section, which shows a table with the following data:

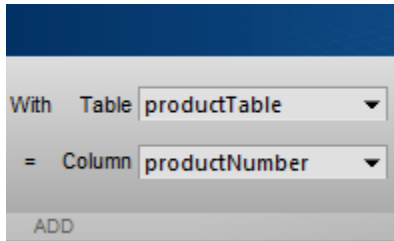
	productNumber	Quantity	Price	inventoryDate
1	1	1700	14.5000	2014-09-23 0...
2	2	1200		9 2014-07-08 2...
3	3	356		17 2014-05-14 0...
4	4	2580		21 2013-06-08 1...
5	5	9000		3 2012-09-14 1...
6	6	4540		8 2013-12-25 1...
7	7	6034		16 2014-08-06 0...
8	8	8350		5 2011-06-18 1...
9	9	2339		13 2011-02-09 1...

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 **Data Browser** pane appears in the left **Table** list. For details about joining tables, see “Join Tables Using Database Explorer App” on page 4-6.

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.

Database Explorer - dbdemo

JOIN

Join Table: inventoryTable With Table: productTable

Column: productNumber = Column: productNumber

INNER JOIN inventoryTable.productNumber=

Inner Full

Remove Join Left Right Close Join

SQL 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)

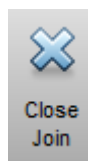
	productNumber	Quantity	Price	inventoryDate
1	9	2339	13	2011-02-09 1...
2	8	8350	5	2011-06-18 1...
3	7	6034	16	2014-08-06 0...
4	2	1200	9	2014-07-08 2...
5	4	2580	21	2013-06-08 1...
6	1	1700	14.5000	2014-09-23 0...
7	5	9000	3	2012-09-14 1...
8	6	4540	8	2013-12-25 1...
9	3	356	17	2014-05-14 0...

Join Diagram

INNER JOIN

inventoryTable productTable

Table Join



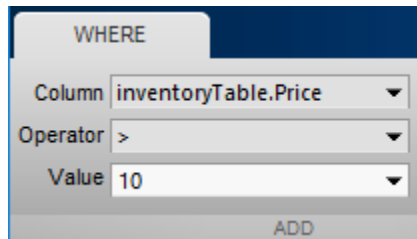
In the **Close** section, click **Close Join** to close the **Join** tab.

In the tree view of the **Data 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 toolbar.

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 image shows a software interface for configuring a WHERE clause filter. It features a tab labeled 'WHERE' at the top. Below the tab are three dropdown menus: 'Column' with 'inventoryTable.Price' selected, 'Operator' with '>' selected, and 'Value' with '10' selected. At the bottom of the panel is an 'ADD' button.

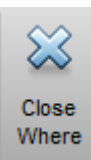
The **SQL Query** and **Data Preview** panes display the updated query results based on the new filter with the **WHERE** condition.

The screenshot shows the Database Explorer interface with a 'WHERE' tab active. The 'Column' is set to 'inventoryTable.Price', the 'Operator' is '>', and the 'Value' is '10'. The SQL query in the main window is:

```
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
```

Below the query, the 'Data Preview (All 6 Rows)' section shows a table with the following data:

	productNumber	Quantity	Price	inventoryDate	productDescription
1	1	1700	14.5000	2014-09-23 0...	Building Blocks
2	3	356	17	2014-05-14 0...	Slinky
3	4	2580	21	2013-06-08 1...	Space Cruiser
4	7	6034	16	2014-08-06 0...	Engine Kit
5	9	2339	13	2011-02-09 1...	Victorian Doll
6	10	723	24	2012-03-14 1...	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
```

Close the database connection by closing the data source tab.

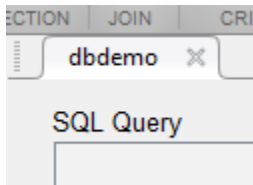
Join Tables Using Left Join and Import Query Results

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.mdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-18.

In the **Data Source** section of the **Database Explorer** tab, click **New Query**. The Connect to a Data Source dialog box opens. Select `dbdemo` from the **Data Source** list. Leave the user name and password blank, and click **Connect**.

The Database Explorer app creates a connection to the Microsoft Access database. The **Data Browser** pane displays the available tables in the database. The data source tab, which is named **dbdemo**, appears to the right of the **Data Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.



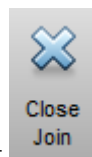
In the **Data Browser** pane, select the **suppliers** table as the first table for the join.

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 **Data Browser** pane appears in the left **Table** list. For details about joining tables, see “Join Tables Using Database Explorer App” on page 4-6.

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.

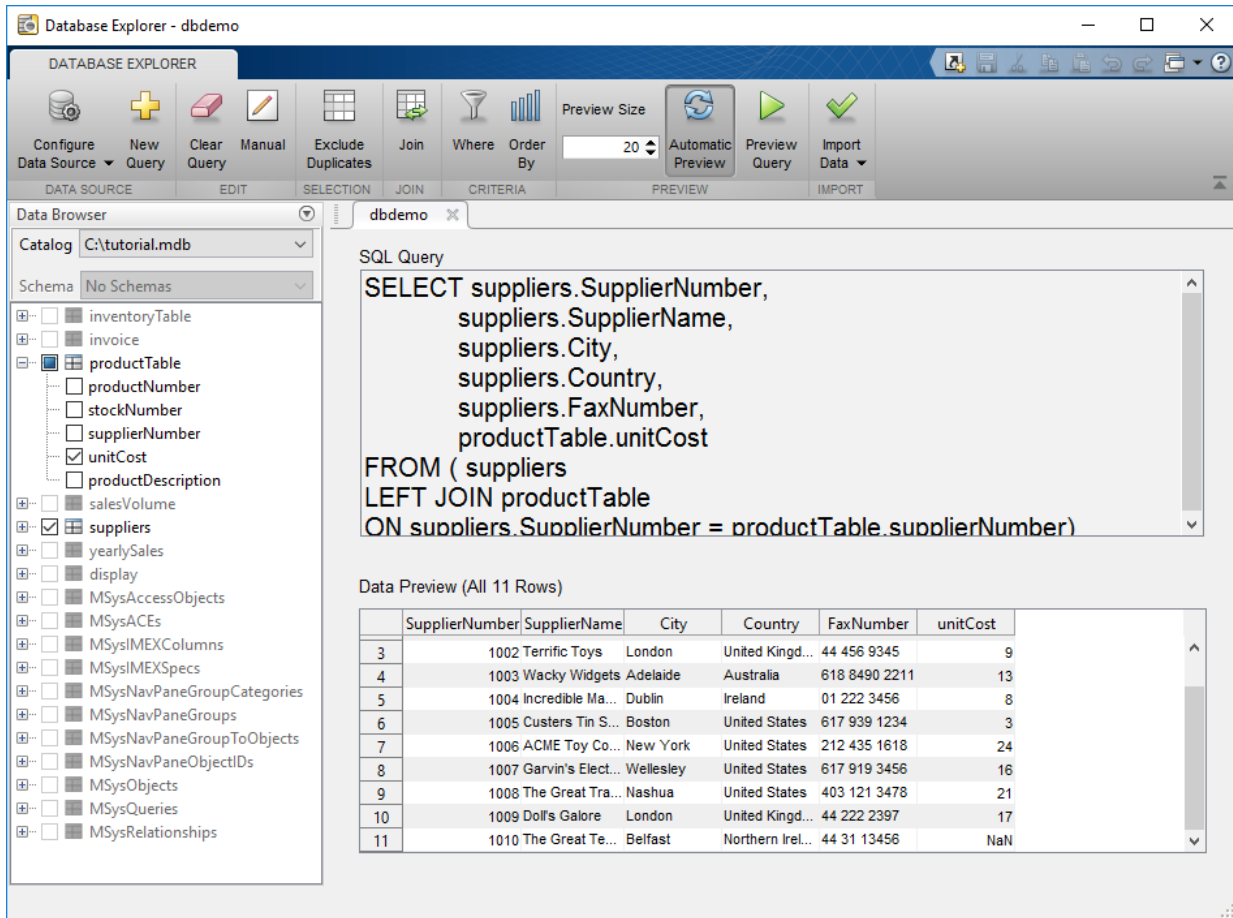
In the **Add** section, click **Add Join**. The Database Explorer app creates an inner join by default. In the **Edit** section, click **Left** to change the join from an inner join to a left join. The **Join Diagram** pane displays a pictorial representation of the join between the selected tables. 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 **Data Browser** pane, select **unitCost** under **productTable**. The **Data Preview** pane updates with a new row.



The NaN value in the **unitCost** column indicates that the corresponding supplier does not supply products.

9	1008	The Great Tra...	Nashua	United States	403 121 3478	21
10	1009	Doll's Galore	London	United Kingd...	44 222 2397	17
11	1010	The Great Te...	Belfast	Northern Irel...	44 31 13456	NaN

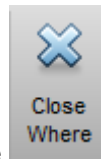
Add filter criteria to the SQL query. In the **Criteria** section, click **Where** to display the **Where** tab in the toolbar.

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 =
```

```
2x6 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

Close the database connection by closing the data source tab.

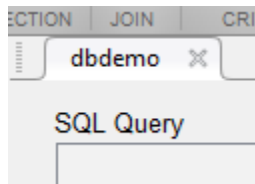
Sort Query Results

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.mdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-18.

In the **Data Source** section of the **Database Explorer** tab, click **New Query**. The Connect to a Data Source dialog box opens. Select `dbdemo` from the **Data Source** list. Leave the user name and password blank, and click **Connect**.

The Database Explorer app creates a connection to the Microsoft Access database. The **Data Browser** pane displays the available tables in the database. The data source tab, which is named `dbdemo`, appears to the right of the **Data Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.

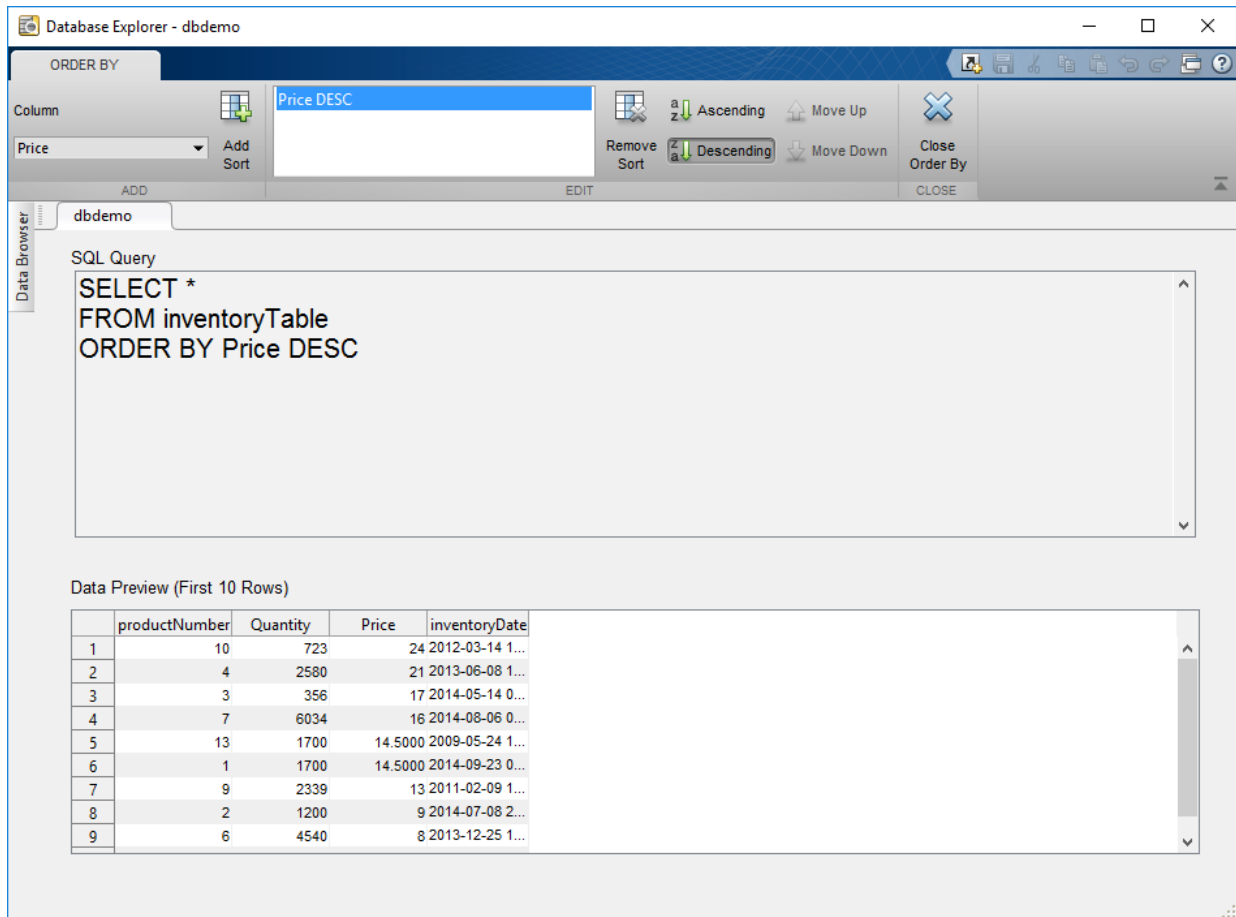


In the **Data Browser** pane, select the `inventoryTable` table. The **SQL Query** pane displays the SQL query that selects all columns and rows from this table. The **Data Preview** pane displays the first 10 rows of the query results.

Sort the results of the SQL query. In the **Criteria** section, click **Order By** to display the **Order By** tab in the toolbar.

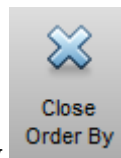
In the **Add** section, in the **Column** list, select the `Price` column. Click **Add Sort**.

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' pane in SQL Server Enterprise Manager. The 'Column' dropdown is set to 'Price' and the 'Sort' dropdown is set to 'Descending'. The 'Data Preview (First 10 Rows)' pane shows the following data:

	productNumber	Quantity	Price	inventoryDate
1	10	723	24	2012-03-14 1...
2	4	2580	21	2013-06-08 1...
3	3	356	17	2014-05-14 0...
4	7	6034	16	2014-08-06 0...
5	13	1700	14.5000	2009-05-24 1...
6	1	1700	14.5000	2014-09-23 0...
7	9	2339	13	2011-02-09 1...
8	2	1200	9	2014-07-08 2...
9	6	4540	8	2013-12-25 1...



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**.

Close the database connection by closing the data source tab.

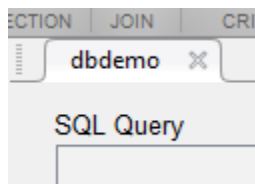
Filter Query Results

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.mdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-18.

In the **Data Source** section of the **Database Explorer** tab, click **New Query**. The Connect to a Data Source dialog box opens. Select `dbdemo` from the **Data Source** list. Leave the user name and password blank, and click **Connect**.

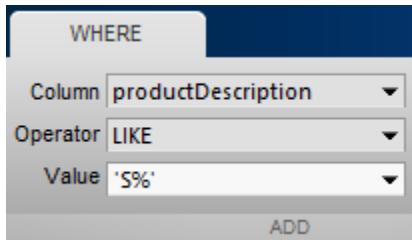
The Database Explorer app creates a connection to the Microsoft Access database. The **Data Browser** pane displays the available tables in the database. The data source tab, which is named `dbdemo`, appears to the right of the **Data Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.



In the **Data 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 toolbar.

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%'`.



The image shows a 'WHERE' filter configuration panel. It has three rows of dropdown menus. The first row is labeled 'Column' and has 'productDescription' selected. The second row is labeled 'Operator' and has 'LIKE' selected. The third row is labeled 'Value' and has ''S%' selected. At the bottom of the panel is an 'ADD' button.

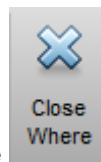
Click **Add Filter**. The **Data Preview** pane displays three rows of data. The product description in each row starts with the letter S.

The screenshot shows the 'Database Explorer - dbdemo' window. At the top, there is a 'WHERE' tab with a filter for 'productDescription LIKE 'S%'' applied to the 'productDescription' column. Below this, the 'SQL Query' section contains the following code:

```
SELECT *
FROM productTable
WHERE productDescription LIKE 'S%'
```

Below the query, the 'Data Preview (All 3 Rows)' section shows a table with the following data:

	productNumber	stockNumber	supplierNumber	unitCost	productDescription
1	4	400339	1008		21 Space Cruiser
2	6	400876	1004		8 Sail Boat
3	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**.

Close the database connection by closing the data source tab.

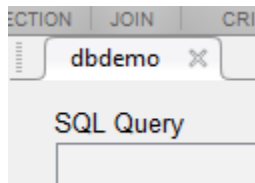
Remove Duplicate Rows from Query Results

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.

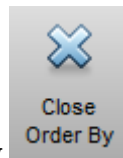
Set up the data source for the `tutorial.mdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-18.

In the **Data Source** section of the **Database Explorer** tab, click **New Query**. The Connect to a Data Source dialog box opens. Select `dbdemo` from the **Data Source** list. Leave the user name and password blank, and click **Connect**.

The Database Explorer app creates a connection to the Microsoft Access database. The **Data Browser** pane displays the available tables in the database. The data source tab, which is named `dbdemo`, appears to the right of the **Data Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.

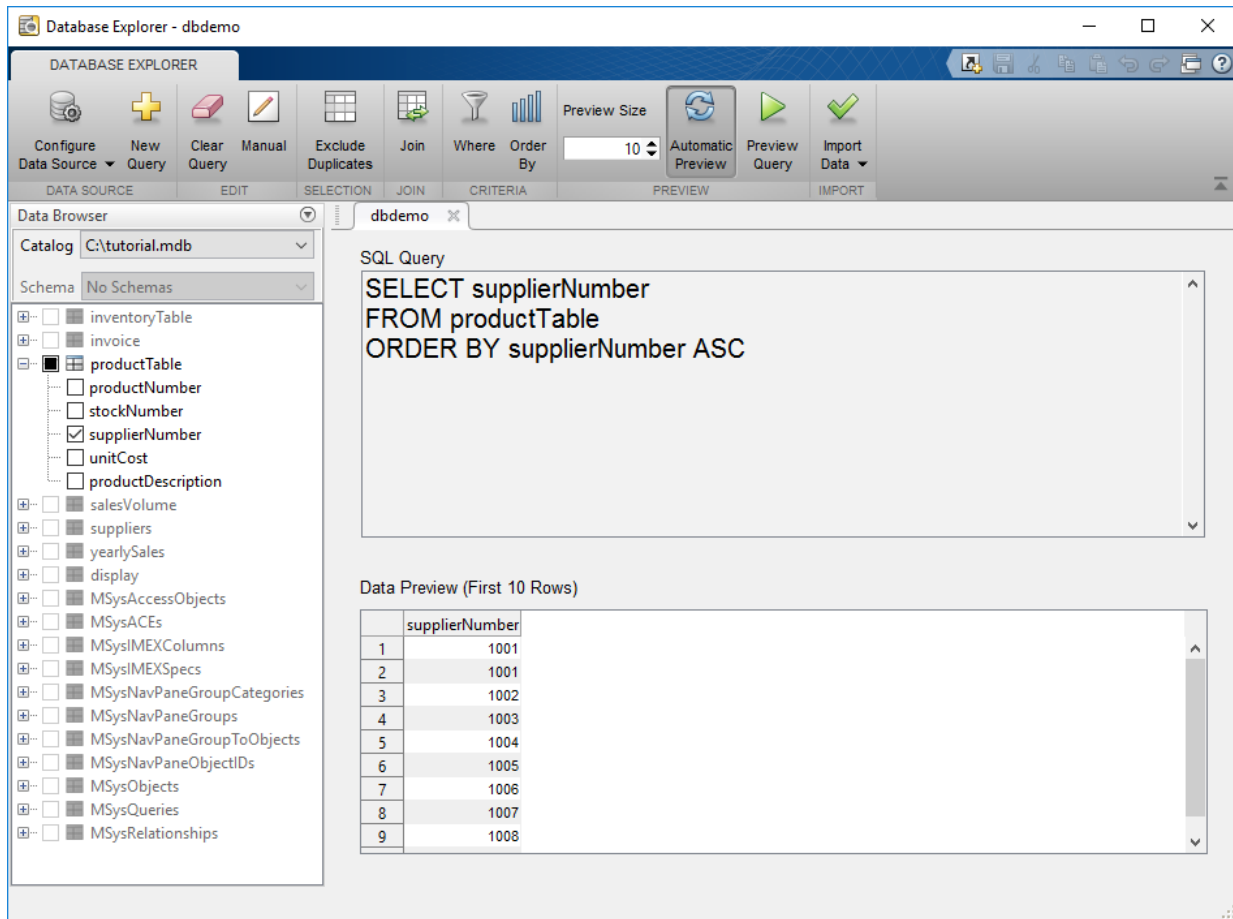


Sort the results of the SQL query. In the **Criteria** section, click **Order By** to display the **Order By** tab in the toolstrip. In the **Add** section, in the **Column** list, select the `supplierNumber` column, and click **Add Sort**.

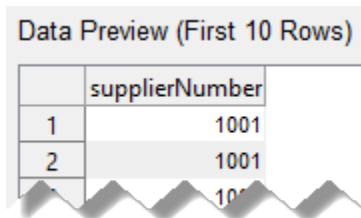


In the **Close** section, click **Close Order By** to close the **Order By** tab.

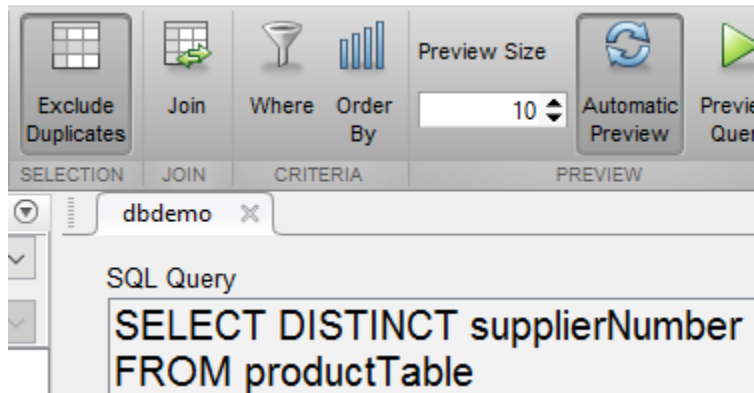
The **Data Preview** pane displays the rows sorted in increasing order, which is the default order.



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.

The screenshot shows the Microsoft Access Database Explorer interface. The 'Data Browser' pane on the left shows the 'productTable' selected, with 'supplierNumber' checked. The 'SQL Query' pane contains the following query:

```
SELECT DISTINCT supplierNumber
FROM productTable
ORDER BY supplierNumber ASC
```

The 'Data Preview (All 9 Rows)' pane shows the following results:

	supplierNumber
1	1001
2	1002
3	1003
4	1004
5	1005
6	1006
7	1007
8	1008
9	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.

Close the database connection by closing the data source tab.

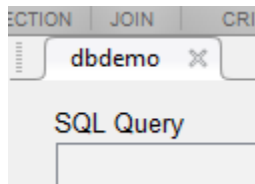
Enter SQL Query Manually

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.mdb` database and name it `dbdemo`. For details, see “Microsoft Access ODBC for Windows” on page 2-18.

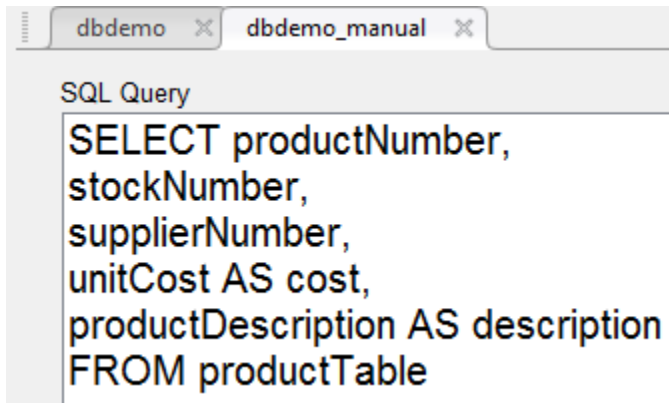
In the **Data Source** section of the **Database Explorer** tab, click **New Query**. The Connect to a Data Source dialog box opens. Select `dbdemo` from the **Data Source** list. Leave the user name and password blank, and click **Connect**.

The Database Explorer app creates a connection to the Microsoft Access database. The **Data Browser** pane displays the available tables in the database. The data source tab, which is named **dbdemo**, appears to the right of the **Data Browser** pane. The data source tab contains two empty panes, **SQL Query** and **Data Preview**.

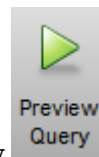


In the **Edit** section, click **Manual**. A new data source tab appears to the right of the **dbdemo** tab with the name **dbdemo_manual**. The suffix `_manual` attached to the tab name indicates that you are entering an SQL query manually.

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.



```
SQL Query
SELECT productNumber,
stockNumber,
supplierNumber,
unitCost AS cost,
productDescription AS description
FROM productTable
```



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 Microsoft Access Database Explorer interface. The main window displays a SQL query in the 'SQL Query' pane and a 'Data Preview (First 10 Rows)' table below it. The 'Data Preview' table contains the following data:

	productNumber	stockNumber	supplierNumber	cost	description
1	9	125970	1003		13 Victorian Doll
2	8	212569	1001		5 Train Set
3	7	389123	1007		16 Engine Kit
4	2	400314	1002		9 Painting Set
5	4	400339	1008		21 Space Cruiser
6	1	400345	1001		14 Building Blocks
7	5	400455	1005		3 Tin Soldier
8	6	400876	1004		8 Sail Boat
9	3	400999	1009		17 Slinky

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.

Close the database connection by closing both the **dbdemo** and **dbdemo_manual** data source tabs.

See Also

Functions

close | database | exec | fetch

Topics

“Connection Options” on page 2-9

“Configuring Driver and Data Source” on page 2-15

“Create SQL Queries Using Database Explorer App” on page 4-2

“Join Tables Using Database Explorer App” on page 4-6

“Data Preview Using Database Explorer App” on page 4-11

“Generate SQL Query and MATLAB Script” on page 4-17

“Modify and Delete Data Sources” on page 4-14

“Database Explorer App Error Messages” on page 3-16

External Websites

SQL Tutorial

Introduced in R2017b

dmd

(To be removed) Construct database metadata object

Note The `dmd` function will be removed in a future release. Use the `sqlfind` function and access the properties of the `connection` object instead.

Syntax

```
dbmeta = dmd(conn)
```

Description

`dbmeta = dmd(conn)` constructs a database metadata object for the database connection `conn`. Use `get` and `supports` to obtain properties of `dbmeta`. Use `dmd` and `get(dbmeta)` to obtain information you need about a database, such as table names required to retrieve data.

Examples

Create a database metadata object `dbmeta` for the database connection `conn` and list its properties:

```
dbmeta = dmd(conn);  
v = get(dbmeta)
```

See Also

[columns](#) | [database](#) | [get](#) | [sqlfind](#) | [supports](#) | [tables](#)

Topics

“Display Database Metadata” on page 5-38

Introduced before R2006a

exec

Package: database.odbc

Execute SQL statement and open cursor

Syntax

```
exec(conn,sqlquery)
```

```
curs = exec(conn,sqlquery)
```

```
curs = exec( ____,Name,Value)
```

```
curs = exec(conn,sqlquery,qTimeOut)
```

Description

`exec(conn,sqlquery)` performs database operations on a 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.

`curs = exec(conn,sqlquery)` creates the `cursor` object after executing the SQL statement `sqlquery` for the database connection `conn`.

`curs = exec(____,Name,Value)` includes any of the input argument combinations in the previous syntaxes and adds options that you specify by one or more `Name,Value` pair arguments. For example, `'MaxRows',10` limits the number of rows to return before SQL query execution to 10 rows.

`curs = exec(conn,sqlquery,qTimeOut)` uses a timeout value `qTimeOut` for SQL query execution.

Examples

Select Data Using Native ODBC Interface

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.

```
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 =
```

```
    []
```

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]
```

With the native ODBC interface, the `Type` property contains the character vector `ODBCCursor Object`. For JDBC connections, this property contains the character vector `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)
```

Select Data Using Timeout Value

Use an ODBC connection with a timeout value 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.

```
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 =
```

```
[]
```

Select all data from the table `productTable` using the `connection` object. Specify a timeout value of 10 seconds. The timeout value specifies the maximum amount of time the `exec` function tries to execute the SQL `SELECT` statement. Assign the SQL `SELECT` statement to the variable `sqlquery`. The cursor object contains the executed SQL query.

```
sqlquery = 'SELECT * FROM productTable';  
qTimeout = 10;  
curs = exec(conn,sqlquery,qTimeout)
```

```
curs =
```

```
    cursor with properties:
```

```
        Data: 0  
        RowLimit: 0  
        SQLQuery: 'SELECT * FROM productTable'  
        Message: []  
        Type: 'ODBCCursor Object'  
        Statement: [1x1 database.internal.ODBCStatementHandle]
```

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)
```

Select Data Using Variable in Query

Use an ODBC connection to import product data from a Microsoft® SQL Server® database into MATLAB® using a variable in the SQL SELECT statement.

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 =
```

```
 []
```

Select data from `productTable` by specifying the product description as a variable. The cursor object contains the executed query. Import the data from the executed query using the `fetch` function.

The SQL SELECT statement uses square brackets to concatenate the two character vectors. To create the pair of single quotation marks that appears in the SQL SELECT statement, specify the pair of four quotation marks around `productdesc`. The outer two marks delineate the next character vector for concatenation. The two inner marks denote a quotation mark inside a character vector.

```
productdesc = 'Painting Set';
sqlquery = ['SELECT * FROM productTable ' ...
           'WHERE productDescription = ' '''' productdesc '''];
curs = exec(conn,sqlquery);
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
 1×5 table
```

<u>productNumber</u>	<u>stockNumber</u>	<u>supplierNumber</u>	<u>unitCost</u>	<u>productDescription</u>
2	4.0031e+05	1002	9	'Painting Set'

Close the cursor object before executing another SQL statement.

```
close(curs)
```

Instead of a variable, use the character vector 'Slinky' to import data.

```
sqlquery = ['SELECT * FROM productTable ' ...  
           'WHERE productDescription = ' ''Slinky'''];  
curs = exec(conn,sqlquery);  
curs = fetch(curs);  
curs.Data
```

```
ans =
```

```
1x5 table
```

<u>productNumber</u>	<u>stockNumber</u>	<u>supplierNumber</u>	<u>unitCost</u>	<u>productDescription</u>
3	4.01e+05	1009	17	'Slinky'

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

Limit Maximum Number of Rows to Return

Use an ODBC connection to import a limited number of rows of 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.

```
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 =
```

```
    []
```

Select all data from productTable. Specify the maximum number of rows to return as two rows.

```
sqlquery = 'SELECT * FROM productTable ORDER BY productNumber';
rowlimit = 2;
curs = exec(conn,sqlquery, 'MaxRows', rowlimit);
```

Display the returned data.

```
curs = fetch(curs);
data = curs.Data
```

```
data =
```

```
2x5 table
```

productNumber	stockNumber	supplierNumber	unitCost	productDescription
1	4.0035e+05	1001	14	'Building Blocks'
2	4.0031e+05	1002	9	'Painting Set'

Determine the highest unit cost in the limited data set.

```
max(data.unitCost)
```

```
ans =
```

14

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)
close(conn)
```

Call Stored Procedure Without Input and Output Arguments

Using a Microsoft SQL Server database, run a stored procedure using the native ODBC database connection `conn`.

Define a stored procedure `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`. Assign the returned cursor object to the variable `curs`.

```
curs = exec(conn, 'create_table')
```



```

curs =
    cursor with properties:
        Data: 0
        RowLimit: 0
        SQLQuery: 'create_table'
        Message: []
        Type: 'ODBCCursor Object'
        Statement: [1x1 database.internal.ODBCStatementHandle]

```

The empty `Message` property means the stored procedure completed successfully.

After you finish working with the cursor object, close it.

```
close(curs)
```

Select Data Using Scrollable Cursor

Use a scrollable cursor object 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.

```

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 =
    []

```

Select all data from the table `productTable` and create a scrollable cursor 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,'CursorType','scrollable')  
  
curs =  
  
    cursor with properties:  
  
        Data: 0  
        RowLimit: 0  
        SQLQuery: 'SELECT * FROM productTable'  
        Message: []  
        Type: 'ODBCCursor Object'  
        Statement: [1x1 database.internal.ODBCStatementHandle]
```

To verify that the `exec` function creates a scrollable cursor, display the hidden `Scrollable` property of the cursor object.

```
curs.Scrollable
```

```
ans =  
  
    logical  
  
    1
```

The `Scrollable` property equals 1 when the database cursor is scrollable.

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)
```

Create Table Using MATLAB® Interface to SQLite

Using the MATLAB® Interface to SQLite, create a table in a new SQLite database file.

Create a 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 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 to the database file, use the `insert` function.

Close the SQLite connection.

```
close(conn)
```

- “Import Data from Databases into MATLAB” on page 5-3
- “Create Queries with Characters and Variables” on page 5-7
- “Call Stored Procedure That Returns Data” on page 5-41
- “Import Data Using MATLAB® Interface to SQLite” on page 5-68
- “Roll Back and Commit Data in Database” on page 5-12
- “Change Database Connection Catalog” on page 5-13
- “Create Table and Add Column” on page 5-14

- “Run Custom Database Function” on page 5-44

Input Arguments

conn — Database connection

connection object | `sqlite` object

Database connection, specified as a `connection` object or `sqlite` object created using the `database` or `sqlite` functions.

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 this 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 `exec` tries to execute the SQL statement, `sqlquery`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

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 a 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 Approaches and Memory Management” on page 5-46.

Data Types: `double`

CursorType — Cursor type

'forward_only' (default) | 'scrollable'

Cursor type, specified as a comma-separated pair consisting of 'CursorType' and one of these values.

Value	Description
'forward_only'	Create a basic cursor.
'scrollable'	Create a scrollable cursor.

For details, see “Using Scrollable Cursors” on page 5-53.

Output Arguments

`curs` — Database cursor

cursor object

Database cursor, returned as a `cursor` object.

Limitations

The name-value pair argument 'MaxRows' has these limitations:

- The native ODBC interface is not supported if you are using Microsoft Access.
- 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 is not constant. To sort records, use the SQL statement `ORDER BY`.
- 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

`exec` executes SQL statements using the command line. To execute SQL statements interactively, use the **Database Explorer** app.

See Also

`close` | `database` | `fetch` | `setdbprefs`

Topics

“Import Data from Databases into MATLAB” on page 5-3

“Create Queries with Characters and Variables” on page 5-7

“Call Stored Procedure That Returns Data” on page 5-41

“Import Data Using MATLAB® Interface to SQLite” on page 5-68

“Roll Back and Commit Data in Database” on page 5-12
“Change Database Connection Catalog” on page 5-13
“Create Table and Add Column” on page 5-14
“Run Custom Database Function” on page 5-44
“Data Import Approaches and Memory Management” on page 5-46
“Using Scrollable Cursors” on page 5-53
“Working with MATLAB Interface to SQLite” on page 2-6
“Data Retrieval Restrictions” on page 1-5

External Websites

SQL Tutorial

Introduced before R2006a

exportedkeys

(To be removed) Retrieve information about exported foreign keys

Note The `exportedkeys` function has been removed.

Syntax

```
e = exportedkeys(dbmeta, 'cata', 'sch')
e = exportedkeys(dbmeta, 'cata', 'sch', 'tab')
```

Description

`e = exportedkeys(dbmeta, 'cata', 'sch')` returns foreign exported key information (that is, information about primary keys that are referenced by other tables) for the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`.

`e = exportedkeys(dbmeta, 'cata', 'sch', 'tab')` returns exported foreign key information for the table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`.

Examples

Get foreign exported key information for the schema `SCOTT` for the database metadata object `dbmeta`.

```
e = exportedkeys(dbmeta, 'orcl', 'SCOTT')
e =
  Columns 1 through 7
  'orcl'  'SCOTT'  'DEPT'  'DEPTNO'  'orcl'...
  'SCOTT'  'EMP'
  Columns 8 through 13
  'DEPTNO'  '1'  'null'  '1'  'FK_DEPTNO'...
  'PK_DEPT'
```


The results show the foreign exported key information.

Column	Description	Value
1	Catalog containing primary key that is exported	null
2	Schema containing primary key that is exported	SCOTT
3	Table containing primary key that is exported	DEPT
4	Column name of primary key that is exported	DEPTNO
5	Catalog that has foreign key	null
6	Schema that has foreign key	SCOTT
7	Table that has foreign key	EMP
8	Foreign key column name, that is the column name that references the primary key in another table	DEPTNO
9	Sequence number within the foreign key	1
10	Update rule, that is, what happens to the foreign key when the primary key updates	null
11	Delete rule, that is, what happens to the foreign key when the primary key is deleted	1
12	Foreign key name	FK_DEPTNO
13	Primary key name that is referenced by foreign key	PK_DEPT

In the schema SCOTT, only one primary key is exported to (referenced by) another table. DEPTNO, the primary key of the table DEPT, is referenced by the field DEPTNO in the table EMP. The referenced table is DEPT and the referencing table is EMP. In the DEPT table, DEPTNO is an exported key. Reciprocally, the DEPTNO field in the table EMP is an imported key.

For a description of codes for update and delete rules, see the `getExportedKeys` property on the Oracle Java Web site:

<http://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html>.

See Also

dmd | get | importedkeys | primarykeys

Topics

“Display Database Metadata” on page 5-38

Introduced before R2006a

fastinsert

Package: database.odbc

(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.

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	Aug
1.2597e+05	1400	1100	981	882	794	752	654	773
2.1257e+05	2400	1721	1414	1191	983	825	731	653
3.8912e+05	1800	1200	890	670	550	450	400	410
4.0031e+05	3000	2400	1800	1500	1200	900	700	650
4.0034e+05	4300	0	2600	1800	1600	1550	895	700
4.0035e+05	5000	3500	2800	2300	1700	1400	1000	900
4.0046e+05	1200	900	800	500	399	345	300	175
4.0088e+05	3000	2400	1500	1500	1300	1100	900	867
4.01e+05	3000	1500	1000	900	750	700	400	350
8.8865e+05	0	900	821	701	689	621	545	423
4.0814e+05	6000	3100	8800	2300	1700	1400	1000	900
2.1046e+05	1800	9700	800	500	3997	349	300	175
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	423

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	Aug
1.2597e+05	1400	1100	981	882	794	752	654	773
2.1257e+05	2400	1721	1414	1191	983	825	731	653
3.8912e+05	1800	1200	890	670	550	450	400	410

4.0031e+05	3000	2400	1800	1500	1200	900	700	650
4.0034e+05	4300	0	2600	1800	1600	1550	895	700
4.0035e+05	5000	3500	2800	2300	1700	1400	1000	900
4.0046e+05	1200	900	800	500	399	345	300	175
4.0088e+05	3000	2400	1500	1500	1300	1100	900	860
4.01e+05	3000	1500	1000	900	750	700	400	350
8.8865e+05	0	900	821	701	689	621	545	420
4.0814e+05	6000	3100	8800	2300	1700	1400	1000	900
2.1046e+05	1800	9700	800	500	3997	349	300	175
4.7082e+05	3100	9400	1540	1500	1350	1190	900	860
5.101e+05	235	1800	1040	900	750	700	400	350
8.9975e+05	123	1700	823	701	689	621	545	420
7.7767e+05	0	350	400	450	250	450	500	510

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 un
3546	'2010-03-01 00:00:00.000'	2	true	[8000×1 un
33116	'2011-05-15 00:00:00.000'	3	true	[8000×1 un
34155	'2011-07-12 00:00:00.000'	4	false	[8000×1 un
34267	'2011-07-22 00:00:00.000'	5	true	[8000×1 un
37197	'2011-09-03 00:00:00.000'	6	true	[8000×1 un
37281	'2011-09-21 00:00:00.000'	7	false	[8000×1 un
41011	'2011-12-12 00:00:00.000'	8	true	[8000×1 un
61178	'2012-01-15 00:00:00.000'	9	false	[8000×1 un
62145	'2012-01-23 00:00:00.000'	10	true	[8000×1 un

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 =
```

```
11x5 table
```

InvoiceNumber	InvoiceDate	ProductNumber	Paid	Receipt
2101	'2010-08-01 00:00:00.000'	1	false	[8000x1 un
3546	'2010-03-01 00:00:00.000'	2	true	[8000x1 un
33116	'2011-05-15 00:00:00.000'	3	true	[8000x1 un
34155	'2011-07-12 00:00:00.000'	4	false	[8000x1 un
34267	'2011-07-22 00:00:00.000'	5	true	[8000x1 un
37197	'2011-09-03 00:00:00.000'	6	true	[8000x1 un
37281	'2011-09-21 00:00:00.000'	7	false	[8000x1 un
41011	'2011-12-12 00:00:00.000'	8	true	[8000x1 un
61178	'2012-01-15 00:00:00.000'	9	false	[8000x1 un
62145	'2012-01-23 00:00:00.000'	10	true	[8000x1 un
2105	'2017-01-04 10:19:42.000'	11	false	'

The last row contains the Boolean data **false**.

Close the database connection.

```
close(conn)
```

- “Export Data to New Record in Database” on page 5-21
- “Export Multiple Records from MATLAB Workspace” on page 5-27
- “Export Data Using Bulk Insert” on page 5-32
- “Replace Existing Data in Database” on page 5-25

- “Roll Back Data After Updating Record” on page 5-18

Input Arguments

conn — Database connection

connection object

Database connection, specified as a `connection` object created with the `database` function.

tablename — Database table name

character vector | string scalar

Database table name, specified as a character vector or string scalar denoting the name of a table in the database.

Example: `'employees'`

Data Types: `char` | `string`

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 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`.

For other differences among these functions, see “Inserting Data Using Command Line” on page 2-148.

See Also

`close` | `commit` | `database` | `exec` | `insert` | `logical` | `rollback` | `select` | `sqlwrite`

Topics

- “Export Data to New Record in Database” on page 5-21
- “Export Multiple Records from MATLAB Workspace” on page 5-27
- “Export Data Using Bulk Insert” on page 5-32
- “Replace Existing Data in Database” on page 5-25
- “Roll Back Data After Updating Record” on page 5-18
- “Inserting Data Using Command Line” on page 2-148
- “Connecting to Database Using Native ODBC Interface” on page 3-19
- “Data Type Support” on page 1-3

External Websites

SQL Tutorial

Introduced before R2006a

fetch

Package: database.odbc

Import data into MATLAB workspace from database cursor or from execution of SQL statement

Syntax

```
curs = fetch(curs)
curs = fetch(curs, rowlimit)
curs = fetch( ___, Name, Value)

results = fetch(conn, sqlquery)
results = fetch(conn, sqlquery, fetchbatchsize)

results = fetch(conn, sqlquery, rowlimit)
```

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 from an executed SQL query.

`curs = fetch(___, Name, Value)` includes any of the input argument combinations in the previous syntaxes and adds scrollable cursor options that you specify by one or more `Name, Value` pair arguments.

You can specify either an absolute or relative position offset. For example, `curs = fetch(curs, 'AbsolutePosition', 5);` imports data using an absolute position offset in a scrollable cursor. While `curs = fetch(curs, 'RelativePosition', 10);` imports data using a relative position offset.

`results = fetch(conn, sqlquery)` returns all rows of data after executing the SQL statement `sqlquery` for the connection or `sqlite` objects. `fetch` imports data in batches.

When you use the connection object as the input argument instead of the cursor object, running the `exec` function is unnecessary.

The `fetch` function imports data from a SQLite database file immediately using a `sqlite` object of the MATLAB interface to SQLite.

`results = fetch(conn, sqlquery, fetchbatchsize)` imports all rows of data in batches of a specified number of rows at a time.

Note This syntax will be removed in a future release. Use the `results = fetch(conn, sqlquery)` syntax instead. The software sets the batch size to a default value automatically.

`results = fetch(conn, sqlquery, rowlimit)` imports the maximum number of rows from an executed SQL query using a `sqlite` object of the MATLAB interface to SQLite.

Examples

Import All Data Using cursor Object

Use the `cursor` object to import product data from a Microsoft® SQL Server® database into MATLAB®. 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, then 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]
```

With the native ODBC interface, the `Type` property of `curs` contains `ODBCCursor Object`. For JDBC connections, the `Type` property contains `Database Cursor Object`.

Display the data in the cursor object property `Data`.

```
curs.Data
```

```
ans =
```

```
    15x5 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'
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 for all products in the table.

```
data = curs.Data;
max(data.unitCost)
```

```
ans =
```

```
24
```

After you finish working with the `cursor` object, close it.

```
close(curs)
```

Import Number of Rows Using `cursor` Object

Use the `cursor` object to import a specific number of rows from a Microsoft® SQL Server® database table into MATLAB®. Then, determine the highest unit cost among the retrieved 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, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
    []
```

Execute the SQL query using the `exec` function and the database connection. Then, use the input argument `rowlimit` to retrieve only the first two rows of data.

```
sqlquery = 'SELECT * FROM productTable';  
curs = exec(conn,sqlquery);  
rowlimit = 2;  
curs = fetch(curs,rowlimit)
```

```
curs =
```

```
    cursor with properties:
```

```
        Data: [2x5 table]  
        RowLimit: 0  
        SQLQuery: 'SELECT * FROM productTable'  
        Message: []  
        Type: 'ODBCursor Object'  
        Statement: [1x1 database.internal.ODBCStatementHandle]  
        Position: 1
```

Display data in the cursor object property `Data`.

```
curs.Data
```

```
ans =
```

```
    2x5 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'

Determine the highest unit cost in the table.

```
data = curs.Data;  
max(data.unitCost)
```

```
ans =
```

```
13
```

After you finish working with the cursor object, close it.

```
close(curs)
```

Import Data Iteratively Using cursor Object

Use a loop with the cursor object to import inventory data from a Microsoft® SQL Server® database table into MATLAB®.

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, '', '');
```

Currently, the data type for imported data is 'table'. Change the data type to 'cellarray' using the `setdbprefs` function.

```
setdbprefs('DataReturnFormat', 'cellarray')
```

Check the database connection. If the `Message` property is empty, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
    []
```

Execute the SQL query using the `exec` function and the database connection. Specify retrieving two rows from `inventoryTable` at a time.

```
sqlquery = 'SELECT inventoryDate FROM inventoryTable';  
curs = exec(conn,sqlquery);  
rowlimit = 2;
```

Use a loop to import data using the `fetch` function

```
while ~strcmp(curs.Data,'No Data')  
    curs = fetch(curs,rowlimit);  
    curs.Data(:)  
end
```

```
ans =
```

```
    2×1 cell array  
  
    '2014-09-23'  
    '2014-07-08'
```

```
ans =
```

```
    2×1 cell array  
  
    '2014-05-14'  
    '2013-06-08'
```

```
ans =
```

```
    2×1 cell array  
  
    '2012-09-14'  
    '2013-12-25'
```

```
ans =  
  
2x1 cell array  
  
'2014-08-06'  
'2011-06-18'
```

```
ans =  
  
2x1 cell array  
  
'2011-02-09'  
'2012-03-14'
```

```
ans =  
  
2x1 cell array  
  
'2012-09-11'  
'2010-10-29'
```

```
ans =  
  
cell  
  
'2009-05-24'
```

```
ans =  
  
cell  
  
'No Data'
```

Set the data type for imported data back to 'table'.

```
setdbprefs('DataReturnFormat','table')
```

After you finish working with the cursor object, close it.


```
close(curs)
```

Import Data with Absolute Position Offset Using Scrollable Cursor

Use a scrollable cursor object to import inventory data from a Microsoft® SQL Server® database table into MATLAB®. Then, determine the highest quantity 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 `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 =
```

```
 []
```

Select all products from the `inventoryTable` table and sort them in ascending order by product number. Create a scrollable cursor object.

```
sqlquery = 'SELECT * FROM inventoryTable ORDER BY productNumber';  
curs = exec(conn,sqlquery, 'CursorType', 'scrollable');
```

Import data in the data set using the absolute position offset 10.

```
curs = fetch(curs, 'AbsolutePosition', 10);
```

Display the imported data.

```
data = curs.Data
```

```
data =
```

```
 4×4 table
```

<u>productNumber</u>	<u>Quantity</u>	<u>Price</u>	<u>inventoryDate</u>
10	723	24	'2012-03-14'
11	567	11	'2012-09-11'
12	1278	22	'2010-10-29'
13	1700	17	'2009-05-24'

After executing `fetch`, the position of the cursor moves after the data set.

Determine the highest quantity in the table.

```
max(data.Quantity)
```

```
ans =
```

```
1700
```

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

Import Data with Row Limit Using Scrollable Cursor

Use a scrollable cursor object and a row limit to import inventory data from a Microsoft® SQL Server® database table into MATLAB®. Then, determine the highest quantity 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 `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 =
```

```
    []
```

Select all products from the table `inventoryTable` and sort them in ascending order by product number. Create a scrollable cursor object.

```
sqlquery = 'SELECT * FROM inventoryTable ORDER BY productNumber';
curs = exec(conn,sqlquery, 'CursorType', 'scrollable');
```

Import data for two products in the middle of the data set. Use the row limit 2 to import data for two rows. Use the absolute position offset 3 to import data starting from the third row in the data set.

```
rowlimit = 2;
curs = fetch(curs,rowlimit, 'AbsolutePosition',3);
```

Display the imported data.

```
data = curs.Data
```

```
data =
```

```
2x4 table
```

productNumber	Quantity	Price	inventoryDate
3	356	17	'2014-05-14'
4	2580	21	'2013-06-08'

Display the position of the scrollable cursor. The position of the cursor stays at the absolute position offset 3.

```
curs.Position
```

```
ans =
```

3

Determine the highest quantity in the table.

```
max(data.Quantity)
```

```
ans =
```

```
2580
```

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)
```

```
close(conn)
```

Import Data with Different Formats Using cursor Object

Use the `cursor` object to import invoice data as a cell array from a Microsoft® SQL Server® database table into MATLAB®. Then, determine the highest invoice number.

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 `Invoice`.

```
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 =
```

```
[]
```

Select the invoice number and paid data from the Invoice table using the `exec` function. The Paid column has data type of Boolean in the database. The cursor object contains the executed SQL query.

```
sqlquery = 'SELECT InvoiceNumber,Paid FROM Invoice';
curs = exec(conn,sqlquery);
```

Currently, the data type for imported data is 'table'. Specify the data type 'cellarray' using the `setdbprefs` function. Import the first five rows of data from the executed SQL query. Display the imported data.

```
setdbprefs('DataReturnFormat','cellarray')
rowlimit = 5;
curs = fetch(curs,rowlimit);
curs.Data
```

```
ans =
```

```
5×2 cell array
```

```
[ 2101]    [0]
[ 3546]    [1]
[33116]    [1]
[34155]    [0]
[34267]    [1]
```

Determine the highest invoice number by accessing the cell array and converting the numeric data to a numeric array using the `cell2mat` function.

```
invoices = curs.Data(1:5,1);
numinvoices = cell2mat(invoices);
max(numinvoices)
```

```
ans =
```

```
34267
```

View the class of the second column in the imported data.

```
class(curs.Data{1,2})
```

```
ans =  
    'logical'
```

Boolean data imports as a `logical` data type in MATLAB®.

Set the data type for imported data back to `'table'`.

```
setdbprefs('DataReturnFormat','table')
```

After you finish working with the `cursor` object, close it.

```
close(curs)
```

Import All Data Using connection Object

Use the `connection` object to import all product data from a Microsoft® SQL Server® database table into MATLAB®. 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, then the connection is successful.

```
conn.Message
```

```
ans =  
    []
```

Import all data from `productTable` using the `connection` object and SQL query. 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	4.0035e+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 for all products in the table.

```
max(results.unitCost)
```

```
ans =
```

```
24
```

Close the database connection.

```
close(conn)
```

Import Data in Batches Using connection Object

Use the `connection` object to import product data in batches from a Microsoft® SQL Server® database table into MATLAB®. 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, then the connection is successful.

```
conn.Message
```

```
ans =
```

```
 []
```

Import all data from `productTable` using the `connection` object and SQL query in batches of five rows at a time. Display the imported data.

```
sqlquery = 'SELECT * FROM productTable';  
fetchbatchsize = 5;  
results = fetch(conn, sqlquery, fetchbatchsize)
```

```
results =
```

```
15x5 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'
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 for all products in the table.

```
max(results.unitCost)
```

```
ans =
```

```
24
```

Close the database connection.

```
close(conn)
```

Import Data Using MATLAB Interface to SQLite

Use the MATLAB® Interface to SQLite to import all data from a table into a SQLite database file. Then, determine the highest unit cost among products in the table.

Create a SQLite connection `conn` to an existing SQLite database file `tutorial.db`. The database file contains the table `productTable`. `conn` is a `sqlite` object.

```
dbfile = 'tutorial.db';
```

```
conn = sqlite(dbfile);
```

Import all data from `productTable` by using the `rowlimit` argument. `results` contains five rows of imported data as a cell array.

```
sqlquery = 'SELECT * FROM productTable';
rowlimit = 5;
results = fetch(conn,sqlquery,rowlimit)
```

```
results =  
  
5x5 cell array  
  
    {[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 unit cost data by looping through the fourth column of the cell array. `data` is a vector that contains numeric unit costs. Find the maximum unit cost.

```
for i = 1:rowlimit  
    data(i) = results{i,4};  
end
```

```
max(data)
```

```
ans =
```

```
int64
```

```
21
```

Close the SQLite connection.

```
close(conn)
```

- “Import Data from Databases into MATLAB” on page 5-3
- “Import Data Using Scrollable Cursor with Relative Position Offset” on page 5-61
- “Retrieve Image Data Types” on page 5-73
- “Display Information About Imported Data” on page 5-50
- “Import Data Using MATLAB® Interface to SQLite” on page 5-68

Input Arguments

curs — Database cursor

cursor object

Database cursor, specified as a cursor object created using the `exec` function.

conn — Database connection

connection object | `sqlite` object

Database connection, specified as a connection object or `sqlite` object created using the `database` or `sqlite` functions.

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 this function. For procedures that return output arguments, use `runstoredprocedure`.

For information about the SQL query language, see the SQL Tutorial.

Data Types: `char` | `string`

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 rows of data.

Data Types: `double`

fetchbatchsize — Fetch batch size

numeric scalar

Fetch batch size, specified as a positive numeric scalar that indicates the number of rows of data to batch at a time. Use `fetchbatchsize` when importing large amounts of data. Retrieving data in batches reduces overall retrieval time.

Data Types: double

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `curs = fetch(curs, 'RelativePosition', 10);`

AbsolutePosition — Absolute position offset

numeric scalar

Absolute position offset, specified as 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, the scalar can be a negative number to signify fetching data from the end of the data set. This name-value pair argument is only available when you create a scrollable cursor object using `exec`. For details, see “Using Scrollable Cursors” on page 5-53.

Example: `'AbsolutePosition', 5`

Data Types: double

RelativePosition — Relative position offset

numeric scalar

Relative position offset, specified as 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, the scalar can be a negative number to signify importing data before the current cursor position in the data set. This name-value pair argument is only available when you create a scrollable cursor object using `exec`. For details, see “Using Scrollable Cursors” on page 5-53.

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 using the `setdbprefs` function.

results — Result data

cell array | table | dataset | structure | numeric matrix

Result data, returned as a cell array, table, dataset array, structure, or numeric matrix as specified by `'DataReturnFormat'` in the `setdbprefs` function. The result data contains all rows of data from the executed SQL statement.

If `conn` is a SQLite connection, then `results` must be a cell array. The cell array contains only one of these data types: `double`, `int64`, or `char`. If NULLs exist in the result data, `fetch` returns an error. To avoid these limitations, connect to the SQLite database file using the JDBC driver. For details, see “Configuring Driver and Data Source” on page 2-15.

Tips

- The order of records in your database does not remain constant. Sort data using the SQL `ORDER BY` command in your `sqlquery` statement.
- 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. Thus, there is always only one `cursor` object created in memory for any of these usages:
 - `curs = fetch(curs)`
 - `fetch(curs)`
 - `curs2 = fetch(curs)`

Alternative Functionality

App

The `fetch` function imports data using the command line. To import data interactively, use the **Database Explorer** app.

See Also

`close` | `database` | `exec` | `fetchmulti` | `setdbprefs`

Topics

“Import Data from Databases into MATLAB” on page 5-3

“Import Data Using Scrollable Cursor with Relative Position Offset” on page 5-61

“Retrieve Image Data Types” on page 5-73

“Display Information About Imported Data” on page 5-50

“Import Data Using MATLAB® Interface to SQLite” on page 5-68

“Using Scrollable Cursors” on page 5-53

“Data Import Approaches and Memory Management” on page 5-46

“Connecting to Database Using Native ODBC Interface” on page 3-19

“Working with MATLAB Interface to SQLite” on page 2-6

External Websites

SQL Tutorial

Introduced before R2006a

fetchmulti

Package: database.odbc

Import data from multiple result sets

Syntax

```
curs = fetchmulti(curs)
```

Description

`curs = fetchmulti(curs)` imports all rows of data from multiple result sets into the `Data` property of the `cursor` object. To create multiple result sets, first execute a SQL query using the `exec` function. The SQL query can contain two or more `SELECT` statements or call a stored procedure that consists of two or more `SELECT` statements. Then, use the `fetchmulti` function to import data in each result set.

Examples

Import Multiple Resultsets

Use the `cursor` object to import inventory and product data from a Microsoft® SQL Server® database using two SQL queries. Then, determine the highest quantity among inventory items.

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 `inventoryTable` and `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 two tables using two `SELECT` statements.

```
sqlquery = 'SELECT * FROM inventoryTable; SELECT * FROM productTable';
```

```
curs = exec(conn,sqlquery);
```

Import data from the two resultsets. The `fetchmulti` function imports data into the `Data` property of the cursor object.

The `Data` property is a cell array consisting of cell arrays, tables, structures, or numeric matrices as specified in the `setdbprefs` function. The data type is the same for all resultsets. Here, `Data` is a cell array of two tables.

The `Data` property contains the data from both resultsets. The first table contains data from the first `SELECT` statement. The second table contains data from the second `SELECT` statement.

```
curs = fetchmulti(curs)
```

```
curs =
```

```
    cursor with properties:
```

```
        Data: {[13×4 table] [15×5 table]}
```

```
        RowLimit: 0
```

```
        SQLQuery: 'SELECT * FROM inventoryTable; SELECT * FROM productTable'
```

```
        Message: []
```

```
        Type: 'ODBCursor Object'
```

```
        Statement: [1×1 database.internal.ODBCStatementHandle]
```

Display data from both tables.

```
inventory = curs.Data{1,1}
```

```
products = curs.Data{1,2}
```


inventory =

13x4 table

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	567	11	'2012-09-11'
12	1278	22	'2010-10-29'
13	1700	17	'2009-05-24'

products =

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 quantity among inventory items.

```
max(inventory.Quantity)
```

```
ans =
```

```
9000
```

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)
```

```
close(conn)
```

- “Call Stored Procedure That Returns Data” on page 5-41
- “Import Data from Databases into MATLAB” on page 5-3
- “Display Information About Imported Data” on page 5-50

Input Arguments

curs — Database cursor

cursor object

Database cursor, specified as a cursor object created using the `exec` function.

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 using the `setdbprefs` function.

See Also

database | exec | fetch | setdbprefs

Topics

“Call Stored Procedure That Returns Data” on page 5-41

“Import Data from Databases into MATLAB” on page 5-3

“Display Information About Imported Data” on page 5-50

“Create Queries with Characters and Variables” on page 5-7

External Websites

SQL Tutorial

Introduced in R2006b

get

Package: database.odbc

(To be removed) Retrieve object properties

Note The `get` function has been removed. Use the `sqlfind` function and access the properties of the `connection` object instead.

The `driver` and `drivermanager` objects have been removed.

Syntax

```
s = get(object)
v = get(object,property)
```

Description

`s = get(object)` returns a structure `s` that contains the object and its corresponding properties.

`v = get(object,property)` returns the value `v` of property for the object.

Examples

Get Database Metadata Object Properties

Retrieve the properties of a database metadata object created using a `connection` object.

Establish an ODBC database connection to a MySQL database with user name `username` and password `pwd`.

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

Construct a database metadata object using the `connection` object.

```
dbmeta = dmd(conn);
```

Retrieve the properties of `dbmeta` and assign them as fields in the structure `v`.

```
v = get(dbmeta)
```

```
v =
  struct with fields:
      AllProceduresAreCallable: 0
      AllTablesAreSelectable: 0
      DataDefinitionCausesTransactionCommit: 1
      DataDefinitionIgnoredInTransactions: 0
      DoesMaxRowSizeIncludeBlobs: 0
      ...
```

Display catalog names in the database.

```
v.Catalogs
```

```
ans =
```

```
  2×1 cell array
    'information_schema'
    'toy_store'
```

Close the database connection.

```
close(conn)
```

Get the AutoCommit Flag Status

Retrieve the `'AutoCommit'` property of the connection object.

Establish an ODBC database connection to a MySQL database with user name `username` and password `pwd`.

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

Check the status of the `'AutoCommit'` property of the connection object.

```
v = get(conn, 'AutoCommit')
```

```
v =  
    1×2 char array  
    'on'
```

Close the database connection.

```
close(conn)
```

- “Display Database Metadata” on page 5-38
- “Display Information About Imported Data” on page 5-50

Input Arguments

object — Database Toolbox object

connection object | cursor object | ...

Database Toolbox object, specified as the following allowable objects:

- connection object, which is created using `database`
- cursor object, which is created using `exec`
- Database metadata object, which is created using `dmd`

Note Support for the database metadata object will be removed in a future release.

property — Property of Database Toolbox object

character vector | string scalar

Property of the Database Toolbox object, specified as a character vector or string scalar.

For connection objects, see this table for the available property names and returned values.

connection Object Property	Description
'AutoCommit'	'on' or 'off', as specified by set. When 'AutoCommit' is set to 'on', the database automatically commits changes to the data. When 'AutoCommit' is set to 'off', the database requires an execution of the SQL COMMIT statement for committing changes to the data.
'Catalogs'	Name of catalogs in the data source. Extract a single catalog name from 'Catalog' for functions such as columns, which accept only a single catalog.
'Driver'	Driver used for a JDBC connection, as specified by database.
'DataSource'	Name of the data source for an ODBC connection or the name of a database for a JDBC connection, as specified by database.
'Message'	Error message returned by database.
'ReadOnly'	'on' if the database is read-only; 'off' if the database is writable.
'LoginTimeout'	Number of seconds that the driver waits while trying to establish a database connection before throwing an error.
'Type'	Object type.
'URL'	For JDBC connections only, the JDBC URL object <code>jdbc:subprotocol:subname</code> , as specified by database.
'UserName'	User name required to connect to a given database, as specified by database.

You cannot use the get function to retrieve the Password property.

For cursor objects, see this table for the 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 returned by fetch, as specified by set.

cursor Object Property	Description
'SQLQuery'	SQL statement for a cursor object, as specified by exec.
'Message'	Error message returned from exec or fetch.
'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 only available for a scrollable cursor. This property behaves differently for native ODBC, JDBC, and different database drivers. This property is read-only.

For database metadata objects, see this table for the available property names and returned values.

Database Metadata Object Property	Description	Example of Value
'Catalogs'	List of database catalogs	{'toystore' 'dbo'}
'DatabaseProductName'	Database vendor name	'ACCESS'
'DatabaseProductVersion'	Database version number	'03.50.0000'
'DriverName'	Name of the JDBC or ODBC driver	'sqlncli11.dll'
'MaxColumnNameLength'	Maximum length of the database column name	64
'MaxColumnsInOrderBy'	Maximum number of database columns for sorting the data	10
'URL'	JDBC database URL for establishing a connection	'jdbc:odbc:dbdemo'

When CatalogName and TableName contain the value {' ' '}, databases do not return metadata for catalog and table names.

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 | cell array | object

Object property value, returned as a character vector, numeric value, cell array, or object.

See Also

close | columns | database | dmd | exec | fetch | getdatasources | rows | set | sqlfind

Topics

“Display Database Metadata” on page 5-38

“Display Information About Imported Data” on page 5-50

Introduced before R2006a

getdatasources

Return names of ODBC and JDBC data sources

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.

Note If you define a JDBC data source with the same name as an existing ODBC data source, the Database Explorer app appends `_JDBC` to the data source name.

See Also

Functions

`database`

Apps

Database Explorer

Topics

“Connecting to Database” on page 2-142

“Configuring Driver and Data Source” on page 2-15

Introduced before R2006a

importedkeys

(To be removed) Return information about imported foreign keys

Note The `importedkeys` function has been removed.

Syntax

```
i = importedkeys(dbmeta, 'cata', 'sch')
i = importedkeys(dbmeta, 'cata', 'sch', 'tab')
```

Description

`i = importedkeys(dbmeta, 'cata', 'sch')` returns foreign imported key information, that is, information about fields that reference primary keys in other tables, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`.

`i = importedkeys(dbmeta, 'cata', 'sch', 'tab')` returns foreign imported key information in the table `tab`. In turn, fields in `tab` reference primary keys in other tables in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`.

Examples

Get foreign key information for the schema `SCOTT` in the catalog `orcl`, for `dbmeta`.

```
i = importedkeys(dbmeta, 'orcl', 'SCOTT')
i =
  Columns 1 through 7
  'orcl'  'SCOTT'  'DEPT'  'DEPTNO'  'orcl'...
  'SCOTT'  'EMP'
  Columns 8 through 13
  'DEPTNO'  '1'  'null'  '1'  'FK_DEPTNO'...
  'PK_DEPT'
```

The results show foreign imported key information as described in the following table.

Column	Description	Value
1	Catalog containing primary key, referenced by foreign imported key	orcl
2	Schema containing primary key, referenced by foreign imported key	SCOTT
3	Table containing primary key, referenced by foreign imported key	DEPT
4	Column name of primary key, referenced by foreign imported key	DEPTNO
5	Catalog that has foreign imported key	orcl
6	Schema that has foreign imported key	SCOTT
7	Table that has foreign imported key	EMP
8	Foreign key column name, that is the column name that references the primary key in another table	DEPTNO
9	Sequence number within foreign key	1
10	Update rule, that is, what happens to the foreign key when the primary key updates	null
11	Delete rule, that is, what happens to the foreign key when the primary key is deleted	1
12	Foreign imported key name	FK_DEPTNO
13	Primary key name in referenced table	PK_DEPT

In the schema SCOTT, there is only one foreign imported key. The table EMP contains a field, DEPTNO, that references the primary key in the DEPT table, the DEPTNO field.

EMP is the referencing table and DEPT is the referenced table.

DEPTNO is a foreign imported key in the EMP table. Reciprocally, the DEPTNO field in the table DEPT is an exported foreign key and the primary key.

For a description of the codes for update and delete rules, see the `getImportedKeys` property on the Oracle Java Web site:

<http://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html>.

See Also

dmd | exportedkeys | get | primarykeys

Topics

“Display Database Metadata” on page 5-38

Introduced before R2006a

indexinfo

(To be removed) Return indices and statistics for database tables

Note The `indexinfo` function has been removed.

Syntax

```
x = indexinfo(dbmeta, 'cata', 'sch', 'tab')
```

Description

`x = indexinfo(dbmeta, 'cata', 'sch', 'tab')` returns indices and statistics for the table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`.

Examples

Get index and statistics information for the table `DEPT` in the schema `SCOTT` of the catalog `orcl`, for `dbmeta`.

```
x = indexinfo(dbmeta, '', 'SCOTT', 'DEPT')
x =
Columns 1 through 8
'orcl' 'SCOTT' 'DEPT' '0' 'null' 'null' '0' '0'
'orcl' 'SCOTT' 'DEPT' '0' 'null' 'PK_DEPT' '1' '1'

Columns 9 through 13
'null' 'null' '4' '1' 'null'
'DEPTNO' 'null' '4' '1' 'null'
```

The results contain two rows, meaning there are two index columns. The statistics for the first index column appear in the following table.

Column	Description	Value
1	Catalog	orcl
2	Schema	SCOTT
3	Table	DEPT
4	Not unique: 0 if index values can be not unique, 1 otherwise	0
5	Index catalog	null
6	Index name	null
7	Index type	0
8	Column sequence number within index	0
9	Column name	null
10	Column sort sequence	null
11	Number of rows in the index table or number of unique values in the index	4
12	Number of pages used for the table or number of pages used for the current index	1
13	Filter condition	null

For details about the index information, see the `getIndexInfo` property on the Oracle Java Web site:

<http://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html>.

See Also

dmd | get | tables

Topics

“Display Database Metadata” on page 5-38

Introduced before R2006a

insert

Package: database.odbc

Add MATLAB data to database tables

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.

Note Support for the connection object will be removed in a future release. Use the `sqlwrite` function instead.

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)
```

Insert Contents of Cell Array

Create an ODBC database connection 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 `yearlySales` with these columns: `Month`, `salesTotal`, and `Revenue`.

Select and display the data from the `yearlySales` table. The cursor object contains the executed query. Import the data from the executed query using the `fetch` function.

```
curs = exec(conn, 'SELECT * FROM yearlySales');
curs = fetch(curs);
curs.Data
```

```
ans =
```

Month	salesTotal	Revenue
'January'	130	1200
'Feb'	25	250

Store the column names of `yearlySales` in a cell array.

```
colnames = {'Month', 'salesTotal', 'Revenue'};
```

Store the data for insertion in a cell array. The data contains `Month` equal to `'March'`, `salesTotal` equal to `$50`, and `Revenue` equal to `$2000`.

```
data = {'March', 50, 2000};
```

Insert the data into `yearlySales`.

```
tablename = 'yearlySales';
insert(conn, tablename, colnames, data)
```

Display the data from `yearlySales` again.

```
curs = exec(conn, 'SELECT * FROM yearlySales');
curs = fetch(curs);
curs.Data
```

```
ans =
```

Month	salesTotal	Revenue
'January'	130	1200
'Feb'	25	250
'March'	50	2000

A new row appears in `yearlySales` with the data from `data`.

After you finish working with the `cursor` object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

Insert Table Record Using MATLAB® Interface to SQLite

Create a table in a new SQLite database file and insert a new row of data into the table.

Create a 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 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)
```

- “Export Data to New Record in Database” on page 5-21
- “Export Multiple Records from MATLAB Workspace” on page 5-27
- “Export Data Using Bulk Insert” on page 5-32
- “Import Data Using MATLAB® Interface to SQLite” on page 5-68
- “Roll Back Data After Updating Record” on page 5-18

Input Arguments

conn — Database connection

connection object | sqlite object

Database connection, specified as a `connection` object or `sqlite` object created using the `database` or `sqlite` functions.

tablename — Database table name

character vector | string scalar

Database table name, specified as a character vector or string scalar denoting the name of a table in the database.

Example: `'employees'`

Data Types: `char` | `string`

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 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`.

For the MATLAB interface to SQLite, use only `insert`. For details, see “Working with MATLAB Interface to SQLite” on page 2-6.

For other differences among these functions, see “Inserting Data Using Command Line” on page 2-148.

See Also

`close` | `commit` | `database` | `fastinsert` | `rollback` | `sqlwrite`

Topics

“Export Data to New Record in Database” on page 5-21

“Export Multiple Records from MATLAB Workspace” on page 5-27

“Export Data Using Bulk Insert” on page 5-32

“Import Data Using MATLAB® Interface to SQLite” on page 5-68

“Roll Back Data After Updating Record” on page 5-18

“Inserting Data Using Command Line” on page 2-148

“Connecting to Database Using Native ODBC Interface” on page 3-19

“Working with MATLAB Interface to SQLite” on page 2-6

“Data Type Support” on page 1-3

External Websites

SQL Tutorial

Introduced before R2006a

isreadonly

(To be removed) Determine if database connection is read-only

Note The `isreadonly` function has been removed. Use the `ReadOnly` property of the `connection` object instead.

Syntax

```
a = isreadonly(conn)
```

Description

`a = isreadonly(conn)` returns 1 if the database connection `conn` is read-only. Otherwise, it returns 0.

Examples

Check whether `conn` is read-only.

```
a = isreadonly(conn)
```

For ODBC connections, you can use the native ODBC interface. For details, see `database`.

The result indicates that the database connection `conn` is read-only:

```
a =  
  1
```

Therefore, you cannot run `datainsert`, `fastinsert`, `insert`, or `update` functions on this database.

See Also

database | isopen

Topics

“Import Data from Databases into MATLAB” on page 5-3

“Export Data to New Record in Database” on page 5-21

“Connecting to Database” on page 2-142

“Connecting to Database Using Native ODBC Interface” on page 3-19

Introduced before R2006a

logintimeout

(To be removed) Set or get time allowed to establish database connection

Note The `logintimeout` function has been removed. Use the `LoginTimeout` name-value pair argument of the database function instead.

Syntax

```
timeout = logintimeout('driver',time)
timeout = logintimeout(time)
timeout = logintimeout('driver')
timeout = logintimeout
```

Description

`timeout = logintimeout('driver',time)` sets the amount of time, in seconds, for a MATLAB session to connect to a database via a given JDBC driver. Use `logintimeout` before running the database function. If the MATLAB session cannot connect to the database within the specified time, it stops trying.

`timeout = logintimeout(time)` sets the amount of time, in seconds, allowed for a MATLAB session to try to connect to a database via an ODBC connection. Use `logintimeout` before running the database function. If the MATLAB session cannot connect within the allowed time, it stops trying.

`timeout = logintimeout('driver')` returns the `time`, in seconds, that was previously specified for the JDBC driver. A returned value of 0 means that the timeout value was not previously set. The MATLAB session stops trying to connect to the database if it is not immediately successful.

`timeout = logintimeout` returns the `time`, in seconds, that you previously specified for an ODBC connection. A returned value of 0 means that the timeout value was not previously set; the MATLAB software session stops trying to make a connection if it is not immediately successful.

Note If you do not specify a value for `logintimeout` and the MATLAB session cannot establish a database connection, your MATLAB session might freeze.

Note Apple Mac platforms do not support `logintimeout`.

Examples

Example 1 — Get Timeout Value for ODBC Connection

View the current connection timeout value.

```
logintimeout
ans =
    0
```

This indicates that you have not specified a timeout value.

Example 2 — Set Timeout Value for ODBC Connection

Set the timeout value to 5 seconds.

```
logintimeout(5)
ans =
    5
```

Example 3 — Get and Set Timeout Value for JDBC Connection

- 1 Check the timeout value for a database connection that is established using an Oracle JDBC driver.

```
logintimeout('oracle.jdbc.driver.OracleDriver')
ans =
    0
```

This indicates that the timeout value is currently 0.

- 2 Set the timeout to 5 seconds.

```
timeout = ...
logintimeout('oracle.jdbc.driver.OracleDriver',5)
```

```
timeout =  
    5
```

- 3** Verify the timeout value.

```
logintimeout('oracle.jdbc.driver.OracleDriver')  
ans =  
    5
```

See Also

database | get | isopen | isreadonly | set

Topics

“Connecting to Database” on page 2-142

“Connecting to Database Using Native ODBC Interface” on page 3-19

Introduced before R2006a

ping

(To be removed) Retrieve status information about database connection

Note The ping function has been removed. Use these connection object properties instead:

- MaxDatabaseConnections
 - DatabaseProductName
 - DatabaseProductVersion
 - DriverName
 - DriverVersion
-

Syntax

```
ping(conn)
```

Description

ping(conn) retrieves the status of the database connection conn.

Examples

Retrieve Status of ODBC Connection

Create an Oracle connection using an ODBC driver. This code assumes that you are connecting a data source named dbname with user name username and password pwd.

```
conn = database(dbname,username,pwd);
```

Retrieve the status of the Oracle connection.

```
ping(conn)

ans =

  struct with fields:

    DatabaseProductName: 'Oracle'
    DatabaseProductVersion: '12.01.0020'
    ODBCDriverName: 'SQORA32.DLL'
    ODBCDriverVersion: '11.02.0004'
    MaxDatabaseConnections: 0
    CurrentUserName: 'username'
    DatabaseURL: ''
    AutoCommitTransactions: 'on'
```

ping returns these fields:

- Database name
- Database version
- JDBC driver name
- JDBC driver version
- Maximum number of database connections allowed
- User name for the current connection
- Database URL

The last field denotes if the current database connection permits automatic commit of transactions.

Close the database connection.

```
close(conn)
```

Retrieve Status of JDBC Connection

Create a Microsoft SQL Server connection using a JDBC driver. For example, the following code assumes that you are connecting a data source named `dbname` with user name `username`, password `pwd`, database server name `sname`, and port number `123456`.


```
conn = database('dbname', 'username', 'pwd', ...  
              'Vendor', 'Microsoft SQL Server', 'Server', 'sname', ...  
              'AuthType', 'Server', 'PortNumber', 123456);
```

Retrieve the status of the Microsoft SQL Server connection.

```
ping(conn)
```

```
ans =
```

```
    DatabaseProductName: 'Microsoft SQL Server'  
    DatabaseProductVersion: '11.00.3000'  
        JDBCDriverName: 'Microsoft JDBC Driver 4.0 for SQL Server'  
        JDBCDriverVersion: '4.0.2206.100'  
    MaxDatabaseConnections: 0  
        CurrentUserName: 'username'  
        DatabaseURL: 'jdbc:sqlserver:...'  
    AutoCommitTransactions: 'True'
```

ping returns these fields:

- Database name
- Database version
- JDBC driver name
- JDBC driver version
- Maximum number of database connections allowed
- User name for the current connection
- Database URL

The last field denotes if the current database connection permits automatic commit of transactions.

Close the database connection.

```
close(conn)
```

- “Import Data from Databases into MATLAB” on page 5-3
- “Export Data to New Record in Database” on page 5-21

Input Arguments

conn — Database connection

connection object

Database connection, specified as a `connection` object created with the `database` function.

Tips

- When you use a `connection` object that is already closed in the `ping` function, the function returns the following error: Invalid connection. Create another connection to your database and try the `ping` function again.

See Also

`close` | `database` | `dmd` | `get` | `isopen` | `set` | `supports`

Topics

“Import Data from Databases into MATLAB” on page 5-3

“Export Data to New Record in Database” on page 5-21

“Connecting to Database” on page 2-142

“Connecting to Database Using Native ODBC Interface” on page 3-19

Introduced before R2006a

primarykeys

(To be removed) Get primary key information for database table or schema

Note The primarykeys function has been removed.

Syntax

```
k = primarykeys(dbmeta, 'cata', 'sch')
k = primarykeys(dbmeta, 'cata', 'sch', 'tab')
```

Description

`k = primarykeys(dbmeta, 'cata', 'sch')` returns primary key information for all tables in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`.

`k = primarykeys(dbmeta, 'cata', 'sch', 'tab')` returns primary key information for the table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`.

Examples

Get primary key information for the DEPT table:

```
k = primarykeys(dbmeta, 'orcl', 'SCOTT', 'DEPT')
k =
  'orcl'   'SCOTT'   'DEPT'   'DEPTNO'   '1'   'PK_DEPT'
```

The results show the primary key information as described in the following table.

Column	Description	Value
1	Catalog	orcl

Column	Description	Value
2	Schema	SCOTT
3	Table	DEPT
4	Column name of primary key	DEPTNO
5	Sequence number within primary key	1
6	Primary key name	PK_DEPT

See Also

dmd | exportedkeys | get | importedkeys

Topics

“Display Database Metadata” on page 5-38

Introduced before R2006a

procedurecolumns

(To be removed) Get stored procedure parameters and result columns of catalogs

Note The procedurecolumns function has been removed.

Syntax

```
pc = procedurecolumns(dbmeta, 'cata', 'sch')
pc = procedurecolumns(dbmeta, 'cata')
```

Description

`pc = procedurecolumns(dbmeta, 'cata', 'sch')` returns the stored procedure parameters and result columns for the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`.

`pc = procedurecolumns(dbmeta, 'cata')` returns stored procedure parameters and result columns for the catalog `cata`, for the database whose database metadata object is `dbmeta`.

Running the stored procedure generates results. One row is returned for each column.

Examples

Get stored procedure parameters for the schema `ORG`, in the catalog `tutorial`, for the database metadata object `dbmeta`:

```
pc = procedurecolumns(dbmeta, 'tutorial', 'ORG')
pc =
Columns 1 through 7
 [1x19 char]  'ORG'  'display'  'Month'  '3'...
 '12'  'TEXT'
 [1x19 char]  'ORG'  'display'  'Day'  '3'...
 '4'  'INTEGER'
```

```

Columns 8 through 13
'50'      '50'      'null'    'null'    '1'      'null'
'50'      '4'       'null'    'null'    '1'      'null'

```

The results show stored procedure parameter and result information. Because two rows of data are returned, there are two columns of data in the results. The results show that running the stored procedure `display` returns the `Month` and `Day` columns.

Following is a full description of the `procedurecolumns` results for the first row (`Month`).

Column	Description	Value for First Row
1	Catalog	'D:\orgdatabase\orcl'
2	Schema	'ORG'
3	Procedure name	'display'
4	Column/parameter name	'MONTH'
5	Column/parameter type	'3'
6	SQL data type	'12'
7	SQL data type name	'TEXT'
8	Precision	'50'
9	Length	'50'
10	Scale	'null'
11	Radix	'null'
12	Nullable	'1'
13	Remarks	'null'

For details about the `procedurecolumns` results, see the `getProcedureColumns` property on the Oracle Java Web site:

<http://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html>.

See Also

dmd | get | procedures

Topics

“Display Database Metadata” on page 5-38

Introduced before R2006a

procedures

(To be removed) Get stored procedures for catalogs

Note The procedures function has been removed.

Syntax

```
p = procedures(dbmeta, 'cata')  
p = procedures(dbmeta, 'cata', 'sch')
```

Description

`p = procedures(dbmeta, 'cata')` returns stored procedures in the catalog `cata` for the database whose database metadata object is `dbmeta`.

`p = procedures(dbmeta, 'cata', 'sch')` returns the stored procedures in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`.

Stored procedures are SQL statements that are saved with the database. Use the `exec` function to run a stored procedure. Specify the stored procedure as the `sqlquery` argument instead of explicitly entering the `sqlquery` statement as the argument.

Examples

Get the names of stored procedures for the catalog `DBA` for the database metadata object `dbmeta`:

```
p = procedures(dbmeta, 'DBA')  
p =  
    'sp_contacts'  
    'sp_customer_list'  
    'sp_customer_products'  
    'sp_product_info'
```



```
'sp_retrieve_contacts'  
'sp_sales_order'
```

Execute the stored procedure `sp_customer_list` for the database connection, and fetch all data:

```
curs = exec(conn, 'sp_customer_list');  
curs = fetch(curs);
```

View the results:

```
curs.Data  
ans =  
[101]    'The Power Group'  
[102]    'AMF Corp.'  
[103]    'Darling Associates'  
[104]    'P.S.C.'  
[105]    'Amo & Sons'  
[106]    'Ralston Inc.'  
[107]    'The Home Club'  
[108]    'Raleigh Co.'  
[109]    'Newton Ent.'  
[110]    'The Pep Squad'
```

See Also

`dmd` | `exec` | `get` | `procedurecolumns`

Topics

“Display Database Metadata” on page 5-38

Introduced before R2006a

querytimeout

Package: database.odbc

Get time specified for SQL queries to succeed

Syntax

```
timeout = querytimeout(curs)
```

Description

`timeout = querytimeout(curs)` returns the amount of time, in seconds, allowed for SQL queries of the open cursor object `curs` to succeed. If a given query cannot complete in the specified time, the toolbox stops trying to perform the query.

The database administrator defines timeout values. If the timeout value is zero, queries must complete immediately.

Examples

Get the current database timeout setting for `curs`.

```
querytimeout(curs)
ans =
    10
```

To create a cursor object using an ODBC connection, you can use the native ODBC interface. For details, see `database`.

Limitations

- This error message displays if a given database does not have a database timeout feature:

[Driver]Optional feature not implemented

- ODBC drivers for Microsoft Access and Oracle do not support querytimeout.

See Also

exec | fetch

Topics

“Display Information About Imported Data” on page 5-50

Introduced before R2006a

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`.

See Also

`commit` | `database` | `datainsert` | `get` | `insert` | `update`

Topics

“Roll Back Data After Updating Record” on page 5-18

“Export Data to New Record in Database” on page 5-21

“Replace Existing Data in Database” on page 5-25

Introduced before R2006a

ROWS

Package: database.odbc

Return number of rows in fetched data set

Syntax

```
numrows = rows(curs)
```

Description

`numrows = rows(curs)` returns the number of rows in the fetched data set `curs`.

Examples

Return Number of Rows in cursor Object

After executing an SQL statement, return the number of rows in the cursor object generated by `fetch`.

Establish connection `conn` to a MySQL database with user name `username` and password `pwd`.

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

Execute a `SELECT` query on the `productTable` for product numbers 1 through 5 inclusive.

```
curs = exec(conn, ['SELECT * FROM productTable'...  
                  ' WHERE productNumber >= 1 AND productNumber <= 5']);
```

`exec` returns the cursor object `curs`.

Fetch the data in `curs`.

```
curs = fetch(curs);
```

The `Data` property of `curs` contains the fetched data from the `SELECT` query.

Return the number of rows in the `Data` property of `curs`.

```
numrows = rows(curs)
```

```
numrows =
```

```
    5
```

Display the rows of data in the `Data` property of `curs`.

```
curs.Data
```

```
ans =
```

```

    [2]    [400314]    [1002]    [ 9]    'Painting Set'
    [4]    [400339]    [1008]    [21]    'Space Cruiser'
    [1]    [400345]    [1001]    [14]    'Building Blocks'
    [5]    [400455]    [1005]    [ 3]    'Tin Soldier'
    [3]    [400999]    [1009]    [17]    'Slinky'
```

After you finish working with the cursor object, close it.

```
close(curs)
```

Close the connection.

```
close(conn)
```

- “Display Information About Imported Data” on page 5-50

Input Arguments

curs — Database cursor

cursor object

Database cursor, specified as a `cursor` object created using the `exec` function.

Output Arguments

numrows — Number of rows in database cursor

numeric scalar

Number of rows in the database cursor, returned as a numeric scalar.

See Also

close | cols | database | exec | fetch | get

Topics

“Display Information About Imported Data” on page 5-50

Introduced before R2006a

runsqlscript

Package: database.odbc

Run SQL script on database

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 =
```

```
    1×2 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
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
'Victorian Doll'	'Wacky Widgets'	'Adelaide'	1400
'Painting Set'	'Terrific Toys'	'London'	3000
'Sail Boat'	'Incredible Machines'	'Dublin'	3000
'Slinky'	'Doll's Galore'	'London'	3000
'Convertible'	'Incredible Machines'	'Dublin'	6000
'Hugsy'	'The Great Teddy Bear Company'	'Belfast'	1800
'Pancakes'	'Aunt Jemimas'	'New York'	3100
'Shawl'	'Indian Export'	'Mumbai'	235
'Snacks'	'Indian Export'	'Mumbai'	123

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 =
```

```
1x2 cursor array with properties:
```

```
Data
RowLimit
SQLQuery
Message
Type
Statement
Position
```

Display the imported data for the second query.

```
results(2).Data
```

```
ans =
```

```
1x6 table
```

productDescription	supplierName	city	Jan_Sales	Feb_Sales	Mar_Sales
'Victorian Doll'	'Wacky Widgets'	'Adelaide'	1400	1100	900

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 =
```

```
1x6 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)
```

- “Import Data from Databases into MATLAB” on page 5-3

Input Arguments

conn — Database connection

connection object

Database connection, specified as a `connection` object created with 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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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 8-264 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.

Definitions

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.

See Also

`close` | `database` | `fetch` | `setdbprefs`

Topics

“Import Data from Databases into MATLAB” on page 5-3

“Configuring Driver and Data Source” on page 2-15

“Generate SQL Query and MATLAB Script” on page 4-17

“Data Import Using Database Explorer App or Command Line” on page 2-145

Introduced in R2012a

runstoredprocedure

Package: database.odbc

Call stored procedure with and without input and output arguments

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 “Connecting to Database” on page 2-142. If you are using the native ODBC interface to connect to your database, use `exec` to call the stored procedure.

Syntax

```
results = runstoredprocedure(conn, sname)
results = runstoredprocedure(conn, sname, inputargs)
results = runstoredprocedure(conn, sname, inputargs, outputtypes)
```

Description

`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 “Connecting to Database” on page 2-142. 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 “Connecting to Database” on page 2-142. 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 “Connecting to Database” on page 2-142. 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 “Connecting to Database” on page 2-142. 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 “Connecting to Database” on page 2-142. 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)
```

- “Call Stored Procedure That Returns Data” on page 5-41

Input Arguments

conn — Database connection

connection object

Database connection, specified as a `connection` object created with the `database` function.

sname — 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 5-41.

`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.

See Also

`close` | `database` | `exec` | `fetch`

Topics

“Call Stored Procedure That Returns Data” on page 5-41

“Connecting to Database” on page 2-142

External Websites

SQL Tutorial

Introduced in R2006b

schemas

(To be removed) Get database schema names

Note The schemas function has been removed. Use the Schemas property of the connection object instead.

Syntax

```
s = schemas(conn)
```

Description

`s = schemas(conn)` retrieves schema names in a database using the database connection `conn`.

Examples

Retrieve Schema Names in the Database

Create a database connection `conn` to the Oracle database using the JDBC driver. Use the Vendor name-value pair argument of `database` to specify a connection to an Oracle database. To connect without Windows authentication, use the `DriverType` name-value pair argument of `database` to specify a connection to the database server by specifying the `thin` value. Here, this code assumes that you are connecting to a database named `dbname` with user name `username` and password `pwd`. This code assumes that you are using the database server named `sname` and port number 123456.

```
conn = database('dbname', 'username', 'pwd', ...  
              'Vendor', 'Oracle', 'DriverType', 'thin', ...  
              'Server', 'sname', 'PortNumber', 123456);
```

Alternatively, use the native ODBC interface for an ODBC connection. For details, see `database`.

Retrieve the schema names in the database named `dbname` using the database connection `conn`.

```
s = schemas(conn)
```

```
s =
```

```
Columns 1 through 4
```

```
'ANONYMOUS'      'APEX_040200'      'APEX_PUBLIC_USER'  'APPQOSSYS'
```

```
Columns 5 through 10
```

```
'AUDSYS'        'CTXSYS'          'DBSNMP'           'DIP'             'DVF'             'DVSYS'
```

```
...
```

`s` returns a cell array of schema names in the Oracle database.

Close the connection.

```
close(conn)
```

- “Display Database Metadata” on page 5-38

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with the database function.

Output Arguments

s — Schema names

cell array

Schema names, returned as a cell array containing the names of the schemas in the database. The contents of `s` that you see depend upon your permission settings in the database.

See Also

catalogs | close | columns | database | tables

Topics

“Display Database Metadata” on page 5-38

Introduced in R2010a

select

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
'Smith'	'Male'	38	'Country General Hospital'	-32768	176
'Johnson'	'Male'	43	'VA Hospital'	69	163
'Williams'	'Female'	38	''	64	131
'Jones'	'Female'	0	'VA Hospital'	67	133
'Broen'	'Female'	49	'Country 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'	-32768	180
'Moore'	'Male'	28	'St Mary's Medical Center'	68	-32768
'Taylor'	'Female'	31	'Country General Hospital'	68	132

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
'Smith'	'Male'	38	'Country General Hospital'	-32768	176
'Johnson'	'Male'	43	'VA Hospital'	69	163
'Williams'	'Female'	38	''	64	131
'Jones'	'Female'	0	'VA Hospital'	67	133
'Broen'	'Female'	49	'Country General Hospital'	64	119

```
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
```

'Smith'	'Male'	38	'Country General Hospital'	-32768	176
'Johnson'	'Male'	43	'VA Hospital'	69	163
'Williams'	'Female'	38	''	64	131
'Jones'	'Female'	0	'VA Hospital'	67	133
'Broen'	'Female'	49	'Country 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'	-32768	180
'Moore'	'Male'	28	'St Mary's Medical Center'	68	-32768
'Taylor'	'Female'	31	'Country General Hospital'	68	132

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
'Smith'	'Male'	38	'Country General Hospital'	-32768	176
'Johnson'	'Male'	43	'VA Hospital'	69	163
'Williams'	'Female'	38	''	64	131
'Jones'	'Female'	0	'VA Hospital'	67	133
'Broen'	'Female'	49	'Country 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'      -32768      180
'Moore'       'Male'      28      'St Mary's Medical Center'      68      -32768
'Taylor'     'Female'    31      'Country General Hospital'      68      132
```

```
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 =

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)
```

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
'Smith'	'Male'	38	'Country General Hospital'	-32768	176
'Johnson'	'Male'	43	'VA Hospital'	69	163
'Williams'	'Female'	38	''	64	131
'Jones'	'Female'	0	'VA Hospital'	67	133
'Broen'	'Female'	49	'Country 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'	-32768	180

```

'Moore'      'Male'      28      'St Mary's Medical Center'      68      -32768
'Taylor'     'Female'    31      'Country General Hospital'      68      132

```

```
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 =
```

```
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)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with 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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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
UNSIGNED INT	0

Database Data Type	Default NULL Value
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 2-145.

See Also

`close` | `count` | `database` | `exec` | `fetch`

Topics

“Data Import Using Database Explorer App or Command Line” on page 2-145

“Data Import Approaches and Memory Management” on page 5-46
“Import Data from Databases into MATLAB” on page 5-3

External Websites

SQL Tutorial

Introduced in R2017a

set

Package: database.odbc

(To be removed) Set properties for database or cursor object

Note The `set` function has been removed. To set database data to read-only and to automatically commit updates, use the `ReadOnly` and `AutoCommit` properties of the `connection` object instead.

The `drivermanager` object has been removed.

Syntax

```
set(object, 'property', value)
set(object)
```

Description

`set(object, 'property', value)` sets the value of *property* to *value* for the specified *object*.

`set(object)` displays all properties for *object*.

Valid values for *object* are:

- “connection Objects” on page 8-298 (Create using `database`.)
- “cursor Objects” on page 8-298 (Create using `exec` or `fetch`.)

Not all databases support all the properties. When you try to set a property that your database does not support, you receive an error message.

For `connection` objects and `cursor` objects, you can use the native ODBC interface with `set`. For details about establishing a connection using the native ODBC interface, see `database`.

connection Objects

Valid values for the *property* and *value* arguments for a connection object are as follows.

Property	Value	Description
'AutoCommit'	'on'	The software writes and automatically commits database data when you run <code>datainsert</code> , <code>fastinsert</code> , <code>insert</code> , or <code>update</code> . You cannot use <code>rollback</code> to reverse this process.
	'off'	The software does not automatically commit database data when you run <code>datainsert</code> , <code>fastinsert</code> , <code>insert</code> , or <code>update</code> . Use <code>rollback</code> to reverse this process. When you are sure that your data is correct, use the <code>commit</code> function to commit it to the database. Alternatively, use <code>exec</code> to roll back or commit data to the database.
'ReadOnly'	'off'	Not read-only; that is, writable
	'on'	Read-only

Note For some databases, if you insert data and close the database connection without committing the data to the database, then the data gets committed automatically. Your database administrator can tell you whether your database behaves this way.

cursor Objects

Valid values for the *property* and *value* arguments for a cursor object are as follows.

Property	Value	Description
'RowLimit'	positive integer	Set the RowLimit for fetch. Specify this property instead of passing RowLimit as an argument to the fetch function. When you define RowLimit for fetch by using set, then fetch behaves differently depending on what type of database you are using.

Examples

Example 1 — Set RowLimit for cursor Object

Establish a JDBC connection to a data source. Run `fetch` to retrieve data from the table `EMP`, and then set the row limit to 5.

```
conn = database('orcl','scott','tiger',...
    'oracle.jdbc.driver.OracleDriver',...
    'jdbc:oracle:thin:@144.212.123.24:1822:');
curs = exec(conn,'SELECT * FROM EMP');
set(curs,'RowLimit',5)
curs = fetch(curs)
curs =
```

cursor with properties:

```
Attributes: []
  Data: {5x8 cell}
DatabaseObject: [1x1 database]
  RowLimit: 5
  SQLQuery: 'SELECT * FROM EMP'
  Message: []
  Type: 'Database Cursor Object'
ResultSet: [1x1 oracle.jdbc.driver.OracleResultSet]
  Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
Statement: [1x1 oracle.jdbc.driver.OracleStatement]
  Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

The `RowLimit` property of `curs` is 5 and the `Data` property is `5x8 cell`, indicating that `fetch` returned five rows of data.

In this example, `RowLimit` limits the maximum number of rows you can retrieve. Therefore, rerunning the `fetch` function returns no data.

Example 2 — Set the AutoCommit Flag to On

Run `datainsert` on a database whose `AutoCommit` flag is set to on.

- 1 Determine the status of the `AutoCommit` flag for the database connection `conn`.

```
get(conn, 'AutoCommit')
```

```
ans =  
off
```

The flag is off.

- 2 Set the flag status to on and verify its value.

```
set(conn, 'AutoCommit', 'on');  
get(conn, 'AutoCommit')
```

```
ans =  
on
```

- 3 Insert a cell array `exdata` into column names `colnames` in the table `Growth`.

```
datainsert(conn, 'Growth', colnames, exdata)
```

The software inserts the data and commits the inserted data to the database.

Example 3 — Set the AutoCommit Flag to Off and Commit Data

Insert and commit data into a database whose `AutoCommit` flag is off.

- 1 First set the `AutoCommit` flag to off for database connection `conn`.

```
set(conn, 'AutoCommit', 'off');
```

- 2 Insert a cell array `exdata` into the column names `colnames` in the table `Avg_Freight_Cost`.

```
datainsert(conn, 'Avg_Freight_Cost', colnames, exdata)
```

- 3 Commit the data to the database.

```
commit(conn)
```

Example 4 — Set the AutoCommit Flag to Off and Roll Back Data

Update data in a database whose AutoCommit flag is off. Then use rollback to roll back the data.

- 1 Set the AutoCommit flag to off for database connection conn.

```
set(conn, 'AutoCommit', 'off');
```

- 2 Update the data in colnames in the Avg_Freight_Weight table, for the record selected by whereclause, with data from the cell array exdata.

```
update(conn, 'Avg_Freight_Weight', colnames, exdata, ...  
       whereclause)
```

The software updates the data in the table but does not commit the data to the database.

- 3 Roll back the data.

```
rollback(conn)
```

The database contains the original data present before running update.

See Also

commit | database | datainsert | exec | fetch | get | logintimeout | rollback | update

Topics

“Roll Back Data After Updating Record” on page 5-18

Introduced before R2006a

setdbprefs

Set preferences for retrieval format, errors, NULLs, and more

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 database preferences are set, they are retained across MATLAB sessions.

`setdbprefs(s)` sets preferences specified in the structure `s` to values that you specify.

Examples

Display Current Values

View the current values of all database preferences

Display all database preference properties and their current values.

```
setdbprefs
```

```
ans =  
  
    struct with fields:  
  
    DataReturnFormat: 'table'  
    ErrorHandling: 'store'  
    NullNumberRead: 'NaN'  
    NullNumberWrite: 'NaN'  
    NullStringRead: 'null'  
    NullStringWrite: 'null'  
    FetchInBatches: 'no'  
    FetchBatchSize: '1000'
```

Change Preference Setting

Set a database preference to another value.

Display the current value of the `NullNumberRead` database preference.

```
setdbprefs('NullNumberRead')
```

```
ans =  
  
    'NaN'
```

Each `NULL` number in the database is read into the MATLAB workspace as `NaN`.

Change the value of this preference to `0`.

```
setdbprefs('NullNumberRead', '0')
```

Each `NULL` number in the database is read into the MATLAB workspace as `0`.

Change DataReturnFormat Preference

Change the way data returns to the MATLAB workspace by altering the database `DataReturnFormat` preference.

Specify that database data be imported into MATLAB cell arrays.

```
setdbprefs('DataReturnFormat', 'cellarray')
```

Establish connection `conn` to a MySQL database with user name `username` and password `pwd`. This database contains the table `producttable` with these columns: `productnumber` and `productdescription`.

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

Import data into the MATLAB workspace.

```
sqlquery = 'SELECT productnumber,productdescription FROM producttable';  
curs = exec(conn,sqlquery);  
curs = fetch(curs,3);  
curs.Data
```

```
ans =
```

```
    [9]    'Victorian Doll'  
    [8]    'Train Set'  
    [7]    'Engine Kit'
```

Resulting data displays as a cell array.

Change the data return format from `cellarray` to `numeric`.

```
setdbprefs('DataReturnFormat','numeric')
```

Import data into the MATLAB workspace.

```
sqlquery = 'SELECT productnumber,productdescription FROM producttable';  
curs = exec(conn,sqlquery);  
curs = fetch(curs,3);  
curs.Data
```

```
ans =
```

```
    9    NaN  
    8    NaN  
    7    NaN
```

In the database, the values for `productDescription` are character strings, as seen in the previous example when `DataReturnFormat` was set to `cellarray`. The `productDescription` values cannot be read when they are imported into the MATLAB workspace using the `numeric` format. Therefore, MATLAB treats these values as `NULL` numbers and assigns them the current value for the `NullNumberRead` preference setting of `NaN`.

Change the data return format to `structure`.

```
setdbprefs('DataReturnFormat','structure')
```

Import data into the MATLAB workspace.

```
sqlquery = 'SELECT productnumber,productdescription FROM producttable';  
curs = exec(conn,sqlquery);  
curs = fetch(curs,3);  
curs.Data
```

```
ans =
```

```
    productnumber: [3x1 double]  
 productdescription: {3x1 cell}
```

Resulting data displays as a structure.

View the contents of the structure `curs.Data` to see the data.

```
curs.Data.productdescription  
curs.Data.productnumber
```

```
ans =
```

```
    'Victorian Doll'  
    'Train Set'  
    'Engine Kit'
```

```
ans =
```

```
    9  
    8  
    7
```

After you finish working with the `cursor` object, close it. Close the database connection.

```
close(curs)  
close(conn)
```

Change Write Format for NULL Numbers

Enable the insertion of a NaN as a NULL in the database by altering the write format setting for NULL numbers.

Establish connection `conn` to a MySQL database with user name `username` and password `pwd`. This database contains the table `inventoryTable` with these columns: `productNumber`, `Quantity`, `Price`, and `inventoryDate`.

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

Specify NaN for the `NullNumberWrite` format.

```
setdbprefs('NullNumberWrite','NaN')
```

Numbers represented as NaN in the MATLAB workspace are exported to databases as NULL.

Select data in the table `inventoryTable`.

```
sqlquery = 'SELECT * FROM inventoryTable';  
curs = exec(conn,sqlquery);  
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...'
```

Specify data `ex_data` to export into `inventoryTable`. The variable `ex_data` contains a NaN. For the inventory date, specify the date as the current moment.

```
ex_data = {24,NaN,30.00,datestr(now,'yyyy-mm-dd HH:MM:SS')};
```

Insert `ex_data` into the database using `fastinsert` with column names: `productNumber`, `Quantity`, `Price`, and `inventoryDate`.

```
colnames = {'productNumber','Quantity','Price','inventoryDate'};
```

```
fastinsert(conn,'inventoryTable',colnames,ex_data)
```

Select data in the table `inventoryTable` to see the last row with NaN data.

```
sqlquery = 'SELECT * FROM inventoryTable';  
curs = exec(conn,sqlquery);  
curs = fetch(curs);  
curs.Data
```

```
ans =
```

```
...
[15]    [1200]    [20.3000]    '2014-10-22 10:52...'
[16]    [1400]    [34.3000]    '1999-12-31 00:00...'
[24]    [ NaN]    [      30]    '2014-10-22 11:19...'
```

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)
close(conn)
```

Specify Error Handling Settings

Change the display of errors in MATLAB by altering the database error handling preferences.

Specify the store format for the `ErrorHandling` preference.

```
setdbprefs('ErrorHandling','store')
```

Database Toolbox stores errors generated by running `database` or `exec` in the `Message` property of the returned connection or cursor object.

Establish connection `conn` to a MySQL database with user name `username` and password `pwd`. This database contains the table `productTable` with the column `productdescription`.

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

The cursor object contains the executed query. Close the cursor object. Fetch data from a closed cursor object.

```
sqlquery = 'SELECT productdescription FROM productTable';
curs = exec(conn,sqlquery);
close(curs)
curs = fetch(curs,3)
```

```
curs =
```

```
cursor with properties:
```

```
Data: 0
```

```
RowLimit: 0
SQLQuery: 'SELECT productdescription FROM productTable'
Message: 'Invalid Cursor: Invalid Cursor'
Type: 'ODBCCursor Object'
Statement: [1x1 database.internal.ODBCStatementHandle]
```

The error generated by this operation appears in the `Message` field.

Specify the report format for the `ErrorHandling` preference.

```
setdbprefs('ErrorHandling','report')
```

With the `ErrorHandling` preference setting set to `report`, errors generated by running `database` or `exec` appear immediately in the Command Window.

The cursor object `curs` contains the executed query. Close the cursor object. Fetch data from a closed cursor object.

```
sqlquery = 'SELECT productdescription FROM productTable';
curs = exec(conn,sqlquery);
close(curs)
curs = fetch(curs,3);
```

```
Error using database.odbc.cursor/fetch (line 54)
Invalid Cursor
```

The error generated by this operation appears immediately in the Command Window.

After you finish working with the cursor object, close it. Close the database connection.

```
close(curs)
close(conn)
```

Change Multiple Settings

Change multiple database preference simultaneously using `setdbprefs`.

Specify that NULL strings are read from the database into a MATLAB matrix of doubles as `'NaN'`.

```
setdbprefs({'NullStringRead','DataReturnFormat'},...
{'NaN','numeric'})
```

For details about another way to change multiple settings, see “Assign Values to Structure” on page 8-309.

Assign Values to Structure

Assign values for specific preferences in a structure to let you change multiple database preferences simultaneously.

Assign values for preferences to fields in the structure `s`.

```
s.DataReturnFormat = 'numeric';
s.NullNumberRead = '0';
s
```

```
s =
```

```
struct with fields:
```

```
    DataReturnFormat: 'numeric'
    NullNumberRead: '0'
```

Set preferences using the values in `s`.

```
setdbprefs(s)
```

Run `setdbprefs` to check your preferences settings.

```
setdbprefs
```

```
ans =
```

```
struct with fields:
```

```
    DataReturnFormat: 'numeric'
    ErrorHandling: 'store'
    NullNumberRead: '0'
    NullNumberWrite: 'NaN'
    NullStringRead: 'null'
    NullStringWrite: 'null'
    FetchInBatches: 'no'
    FetchBatchSize: '1000'
```

Return Values to Structure

Capture all preferences and their values in a structure.

Assign values for all preferences to `s`.

```
s = setdbprefs  
  
s =  
  
    struct with fields:  
  
    DataReturnFormat: 'numeric'  
    ErrorHandling:   'store'  
    NullNumberRead:  '0'  
    NullNumberWrite: 'NaN'  
    NullStringRead:  'null'  
    NullStringWrite: 'null'  
    FetchInBatches:  'no'  
    FetchBatchSize:  '1000'
```

Use the MATLAB tab completion feature when obtaining the value for a preference.

```
s.D
```

Press the **Tab** key, and then **Enter**. MATLAB completes the field and displays the value.

```
s.DataReturnFormat
```

```
ans =  
  
    'numeric'
```

Save Preferences

You can save your preferences to a 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
```

Later, load the data and restore the preferences.

```
load ImportDataPrefs.mat
setdbprefs(ImportData)
```

- “Import Data from Databases into MATLAB” on page 5-3

Input Arguments

preference — Database preference

character vector | cell array

Database preference, specified as a character vector. 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.

- 'DataReturnFormat' — Format for data to import into the MATLAB workspace using the preference values listed here. Set the format based on the type of data being retrieved, memory considerations, and your preferred method of working with retrieved data. For example, to specify the format as a table, enter `setdbprefs('DataReturnFormat','table')`.

Note These preferences will be removed in a future release.

Preference Setting	Use This Instead
<code>setdbprefs('DataReturnFormat','dataset')</code>	<code>setdbprefs('DataReturnFormat','table')</code>
<code>setdbprefs('DataReturnFormat','numeric')</code>	
<code>setdbprefs('DataReturnFormat','cellarray')</code>	
<code>setdbprefs('DataReturnFormat','structure')</code>	

Allowable Values	Description
'cellarray' (default)	Import nonnumeric data into MATLAB cell arrays.

Allowable Values	Description
'table'	Import data into a MATLAB table. Use for all data types. Facilitates working with returned columns.
'dataset'	Import data into a MATLAB dataset array. Use for all data types. Facilitates working with returned columns. This value requires Statistics and Machine Learning Toolbox™.
'numeric'	Import data into a MATLAB matrix of doubles. Nonnumeric data types are considered NULL and appear as specified in the 'NullNumberRead' property. Use only when data to retrieve is in numeric format, or when nonnumeric data to retrieve is not relevant.
'structure'	Import data into a MATLAB structure. Use for all data types. Facilitates working with returned columns.

- 'ErrorHandling' — Specify how to handle errors when importing data using the preference values listed here. Set this parameter before you run `database` or `exec`. For example, to specify storing errors in the `Message` field of the returned connection object, enter `setdbprefs('ErrorHandling','store')`.

Note The preference setting `setdbprefs('ErrorHandling','store')` will be removed in a future release. Use `setdbprefs('ErrorHandling','report')` instead.

Allowable Values	Description
'store' (default)	Store errors from running <code>database</code> in the <code>Message</code> field of the returned connection object. Store errors from running <code>exec</code> in the <code>Message</code> property of the returned cursor object.

Allowable Values	Description
'report'	Immediately display errors from running database or exec in the Command Window.

- **NULL data** — Specify how to import or export NULL data in the MATLAB workspace or the database using the preference values listed here. For example, to import data and display all NULL numbers in the database as a 0 in the MATLAB workspace, enter `setdbprefs('NullNumberRead','0')`.

Note These preferences will be removed in a future release.

Preference Setting	Use This Instead
<code>setdbprefs('NullNumberWrite','NaN')</code>	<code>sqlwrite</code>
<code>setdbprefs('NullStringWrite','null')</code>	<code>sqlwrite</code>
<code>setdbprefs('NullNumberRead','0')</code>	Default value NaN for numeric NULL values

Database Preference	Allowable Values	Description
'NullNumberRead'	Character vector; for example, '0'	<p>How NULL numbers appear after being imported from a database into the MATLAB workspace. NaN is the default value.</p> <ul style="list-style-type: none"> • If 'DataReturnFormat' is set to 'numeric', then values such as 'NULL' cannot be set. • If 'DataReturnFormat' is set to 'cellarray', then numbers appear as NaN and not as empty brackets. <p>Set this parameter before running fetch.</p>
'NullNumberWrite'	Character vector; for example, 'NaN' (default)	<p>How numbers appear in the database after being exported from the MATLAB workspace to a database.</p> <p>Regardless of the value of 'NullNumberWrite', a NULL value is always written to the database when you input [] or NaN for a numeric data type.</p>

Database Preference	Allowable Values	Description
'NullStringRead'	Character vector; for example, 'null' (default)	How NULL strings appear after being imported from a database into the MATLAB workspace. Set this parameter before running <code>fetch</code> .
'NullStringWrite'	Character vector; for example, 'null' (default)	Specify the character vector that represents a NULL string in a database after exporting it from the MATLAB workspace to the database. For character vector inputs, a NULL value is written to the database only when the input value matches the value of 'NullStringWrite'.

- Fetching data — Specify how to import data using the preference values listed here. Control the number of rows that are imported from the database at a time. For example, to automate fetching data in batches, enter `setdbprefs('FetchInBatches','yes')`.

Note These database preferences will be removed in a future release. The software sets the batch size to a default value and performs batching by default.

Database Preference	Allowable Values	Description
'FetchInBatches'	'yes' or 'no' (default)	Automates fetching in batches for large data sets where you can run into Java heap memory errors in MATLAB. When the value is 'yes', <code>fetch</code> and <code>runsqlscript</code> import the data in batches in size of 'FetchBatchSize'. For an example, see <code>fetch</code> .
'FetchBatchSize'	Input numeric value, default value is '1000'. Supported values are 1000 through 1000000.	Automates fetching in batches for large data sets when used with 'FetchInBatches'. When the value of 'FetchInBatches' is 'yes', <code>fetch</code> and <code>runsqlscript</code> import the data in batches in size of 'FetchBatchSize'. For an example, see <code>fetch</code> .

Example: 'DataReturnFormat'

Example: {'DataReturnFormat';'NullStringRead'}

Data Types: char

value — Database preference value

character vector | cell array

Database preference value, specified as a character vector. 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. For allowable values, see the tables in preference.

Example: 'NaN'

Example: {'numeric'; 'NaN'}

Data Types: char

s — Database preferences

structure

Database preferences, specified as a structure to include all the database preferences that 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 MATLAB toolstrip and click **Database Toolbox**. Enter values for each database preference.

See Also

clear | close | database | exec | fastinsert | fetch | getdatasources

Topics

“Import Data from Databases into MATLAB” on page 5-3

“Working with Database Toolbox Preferences” on page 2-154

Introduced before R2006a

splitsqlquery

Package: database.odbc

Split SQL query using paging

Syntax

```
querybasket = splitsqlquery(conn,sqlquery)
querybasket = splitsqlquery(conn,sqlquery,'SplitSize',splitsize)
```

Description

`querybasket = splitsqlquery(conn,sqlquery)` splits a 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. Here, 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 using the `fetch` function.

To run this example, ensure that the current folder contains the JDBC driver file `sqljdbc4.jar`. To find this file, see “Microsoft SQL Server JDBC for Windows” on page 2-32.

Add the JDBC driver file to the Java® class path.

```
javaaddpath 'sqljdbc4.jar';
```

Connect to the Microsoft® SQL Server® database using the database name `toy_store`, database server `dbtb04`, port number `54317`, and Windows® authentication.

```
conn = database('toy_store', '', '', 'Vendor', 'Microsoft SQL Server', ...
               'Server', 'dbtb04', 'PortNumber', 54317, 'AuthType', 'Windows');
```

Define the SQL query. Here, 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 =
```

```
    2×1 string array
```

```
    " SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 0 ROWS FETCH NEXT 100000 ROWS ONLY;"
    " SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 100000 ROWS FETCH NEXT 100000 ROWS ONLY;"
```

The query basket contains the page queries in a string array. The `splitsqlquery` function splits these queries using the default number of rows (100,000).

Define the `airlinesdata` variable.

```
airlinesdata = [];
```

Define the minimum arrival delay `minArrDelay` variable.

```
minArrDelay = [];
```

Use a `for` loop to execute the SQL page queries in `querybasket` and import the data in chunks. Execute SQL page queries in the query basket and import large data using the `fetch` function. Find the local minimum arrival delay for a chunk. 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. Here, access the database using a parallel pool.

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.

The definition of large data can vary. Performance for importing large data depends on the SQL query, amount of data, machine specifications, and type of data analysis. To manage the performance, use the `splitsize` input argument of the `splitsqlquery` function.

To run this example, ensure that the current folder contains the JDBC driver file `sqljdbc4.jar`. To find this file, see “Microsoft SQL Server JDBC for Windows” on page 2-32.

If you have a MATLAB® Distributed Computing Server™ license, then you can replace the name of the cluster in the `parpool` function with the cluster profile of your choice.

Add the JDBC driver file to the Java® class path.

```
javaaddpath 'sqljdbc4.jar';
```

Connect to the Microsoft® SQL Server® database using the database name `toy_store`, database server `dbtb04`, port number `54317`, and Windows® authentication.

```
conn = database('toy_store',' ','Vendor','Microsoft SQL Server', ...
    'Server','dbtb04','PortNumber',54317,'AuthType','Windows');
```

Define the SQL query. Here, 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. Use a split size of 10,000 rows.

```
splitsize = 10000;
querybasket = splitsqlquery(conn,sqlquery,'SplitSize',splitsize)
```

```
querybasket =
```

```
13×1 string array
```

```
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 0 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 10000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 20000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 30000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 40000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 50000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 60000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 70000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 80000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 90000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 100000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 110000 ROWS FETCH NEXT 10000 ROWS ONLY;"
" SELECT * FROM (SELECT * FROM airlinesmall) temp ORDER BY 1 OFFSET 120000 ROWS FETCH NEXT 10000 ROWS ONLY;"
```

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 parallel pool using the local profile.

```
p = parpool('local');
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 6 workers.
```

Attach the required JDBC driver file `sqljdbc4.jar`.

```
addAttachedFiles(p,{'sqljdbc4.jar'});
```

Define the `airlinesdata` variable.

```
airlinesdata = [];
```

Define the minimum arrival delay `minArrDelay` variable.

```
minArrDelay = [];
```

Add the JDBC driver file to the Java® class path, and build the connection once for each worker.

```
parfevalOnAll(@javaaddpath,0,'sqljdbc4.jar');
c = parallel.pool.Constant( ...
    @() database('toy_store','','Vendor','Microsoft SQL Server', ...
        'Server','dbtb04','PortNumber',54317,'AuthType', ...
        'Windows'),@close);
```

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.

```
p.delete;
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with 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`

splitsize — SQL query split size

100000 (default) | numeric scalar

SQL query split size, specified as a numeric scalar. Specify this number to split a 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.

See Also

`addAttachedFiles` | `close` | `database` | `fetch` | `javaaddpath` | `parallel.pool.Constant` | `parfevalOnAll` | `parpool`

Topics

“Analyze Large Data in Database Using Tall Arrays”

External Websites

SQL Tutorial

Introduced in R2017b

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 “Working with MATLAB Interface to SQLite” on page 2-6.

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.

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.

Data Types: `logical`

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`

Object Functions

`insert` Add MATLAB data to database tables
`exec` Execute SQL statement and open cursor
`fetch` Import data into MATLAB workspace from database cursor or from execution of SQL statement
`close` Close and invalidate database and driver resource utilizer

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\Bdoc18a_815039_17936\ib0BF173\3\tp081482b9\ex96650978\tutorial.db'  
    IsOpen: 1
```



```
IsReadOnly: 0
```

`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\Bdoc18a_815039_17936\ib0BF173\4\tpd9eec1af\database-ex6195242'
      IsOpen: 1
    IsReadOnly: 0
```

`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, you can create a table using the `exec` function, and then insert data using the `insert` 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\Bdoc18a_815039_17936\ib0BF173\3\tp081482b9\ex41829813\tutorial.db'
      IsOpen: 1
  IsReadOnly: 1
```

`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.

The `sqlite` object provides limited Database Toolbox functionality. For full functionality, create a database connection to the SQLite database file using the JDBC driver. To use the JDBC driver, close the SQLite connection and create a database connection using the URL string. For details, see these links depending on your platform:

- “SQLite JDBC for Windows” on page 2-77
- “SQLite JDBC for macOS” on page 2-127
- “SQLite JDBC for Linux” on page 2-133

See Also

Topics

“Access Relational Database Data in MATLAB” on page 2-3

“Working with MATLAB Interface to SQLite” on page 2-6

“Import Data Using MATLAB® Interface to SQLite” on page 5-68

“Configuring Driver and Data Source” on page 2-15

Introduced in R2016a

sql2native

(To be removed) Convert JDBC SQL grammar to SQL grammar native to system

Note The sql2native function has been removed.

Syntax

```
n = sql2native(conn, 'sqlquery')
```

Description

`n = sql2native(conn, 'sqlquery')` converts the SQL statement `sqlquery` from JDBC SQL grammar into the database system's native SQL grammar for the connection `conn`. The native SQL statement is assigned to `n`.

See Also

database

Topics

“Data Import Using Database Explorer App or Command Line” on page 2-145

Introduced before R2006a

supports

(To be removed) Detect whether property is supported by database metadata object

Note The supports function has been removed.

Syntax

```
a = supports(dbmeta)
a = supports(dbmeta, 'property')
```

Description

`a = supports(dbmeta)` returns a structure that contains the properties of `dbmeta` and its property values, 1 or 0. A value of 1 indicates that the property is supported, and 0 indicates that the property is not supported.

`a = supports(dbmeta, 'property')` returns 1 or 0 for the `property` field of `dbmeta`. A value of 1 indicates that the property is supported, and 0 indicates that the property is not supported.

Examples

- 1 Check if `dbmeta` supports group-by clauses.

```
a = supports(dbmeta, 'GroupBy')
a =
    1
```

- 2 View the value of all properties of `dbmeta`.

```
a = supports(dbmeta)
```

The returned result is a list of properties and their values.

- 3 After creating `a` using the `supports` function, you can access the value of any property in `a`. Display the `GroupBy` property by running:

```
a.GroupBy  
a =  
  1
```

See Also

database | dmd | get | ping

Topics

“Display Database Metadata” on page 5-38

Introduced before R2006a

tableprivileges

(To be removed) Return database table privileges

Note The `tableprivileges` function will be removed in a future release. Use the `sqlfind` function and access the properties of the connection object instead.

Syntax

```
tp = tableprivileges(dbmeta, 'cata')
tp = tableprivileges(dbmeta, 'cata', 'sch')
tp = tableprivileges(dbmeta, 'cata', 'sch', 'tab')
```

Description

`tp = tableprivileges(dbmeta, 'cata')` returns a list of table privileges for all tables in the catalog `cata` for the database whose database metadata object is `dbmeta` resulting from a connection object.

`tp = tableprivileges(dbmeta, 'cata', 'sch')` returns a list of table privileges for all tables in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta` resulting from a connection object.

`tp = tableprivileges(dbmeta, 'cata', 'sch', 'tab')` returns a list of privileges for the table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta` resulting from a connection object.

Examples

Get table privileges for the `builds` table in the schema `geck` for the catalog `msdb`, for the database metadata object `dbmeta`.

```
tp = tableprivileges(dbmeta, 'msdb', 'geck', 'builds')
tp =
```

'DELETE' 'INSERT' 'REFERENCES' ...
'SELECT' 'UPDATE'

See Also

dmd | get | sqlfind | tables

Topics

“Display Database Metadata” on page 5-38

Introduced before R2006a

tables

Package: database.odbc

(To be removed) Return database table names

Note The `tables` function will be removed in a future release. Use the `sqlfind` function and access the properties of the `connection` object instead.

Syntax

```
t = tables(conn, catalog)
t = tables(conn, catalog, schema)

t = tables(dbmeta, catalog)
t = tables(dbmeta, catalog, schema)
```

Description

`t = tables(conn, catalog)` returns a list of all table names and table types for all schemas in the specified catalog named `catalog`.

Note The syntax `tables(conn)` has been removed. Use syntaxes with at least two input arguments instead.

`t = tables(conn, catalog, schema)` returns a list of all table names and table types in the specified catalog named `catalog` and schema named `schema`.

`t = tables(dbmeta, catalog)` returns a list of all table names and table types in the specified catalog named `catalog` using the database metadata object `dbmeta`.

`t = tables(dbmeta, catalog, schema)` returns a list of all table names and table types in the specified catalog named `catalog` and schema named `schema`.

Examples

Retrieve Table List for Catalog Using Database Connection

Create a database connection `conn` using the native ODBC interface to the Microsoft SQL Server database. 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');
```

Retrieve the list of all table names and table types in the catalog using `conn`. Here, this code assumes that the database contains the catalog name `toy_store`.

```
catalog = 'toy_store';
```

```
t = tables(conn, catalog)
```

```
t =
```

```
    'productTable'           'TABLE'  
    'salesVolume'          'TABLE'  
    'COLUMNS'              'VIEW'  
    ...
```

`t` returns a cell array with the table names in the first column and the table types in the second column.

Close the database connection `conn`.

```
close(conn)
```

Retrieve Table List for Catalog and Schema Using Database Connection

Create a database connection `conn` using the native ODBC interface to the Microsoft SQL Server database. 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');
```

Retrieve the list of all table names and table types in the catalog and schema using `conn`. Here, this code assumes that the database contains the catalog name `toy_store` and schema name `sch`.

```
catalog = 'toy_store';
schema = 'sch';

t = tables(conn,catalog,schema)

t =

    'productTable'          'TABLE'
    'salesVolume'          'TABLE'
    'suppliers'             'TABLE'
    ...
```

`t` returns a cell array with the table names in the first column and the table types in the second column.

Close the database connection `conn`.

```
close(conn)
```

Retrieve Table List for Catalog Using Database Metadata Object

Create a database connection `conn`. This code uses database name `dbname`, user name `username`, password `pwd`, database server name `sname`, and port number 123456 to connect to a Microsoft SQL Server database.

```
conn = database('dbname','username','pwd', ...
    'Vendor','Microsoft SQL Server', ...
    'Server','sname', ...
    'PortNumber',123456);
```

Create the database metadata object `dbmeta` using `conn`.

```
dbmeta = dmd(conn);
```

Retrieve the list of all table names and table types in the catalog using `dbmeta`. Here, this code assumes that the database contains the catalog name `toy_store`.

```
catalog = 'toy_store';  
  
t = tables(dbmeta,catalog)  
  
t =  
  
    'productTable'          'TABLE'  
    'salesVolume'         'TABLE'  
    'suppliers'            'TABLE'  
    ...
```

t returns a cell array with the table names in the first column and the table types in the second column.

Close the database connection conn.

```
close(conn)
```

Retrieve Table List for Catalog and Schema Using Database Metadata Object

Create a database connection conn. This code uses database name dbname, user name username, password pwd, database server name sname, and port number 123456 to connect to a Microsoft SQL Server database.

```
conn = database('dbname','username','pwd', ...  
    'Vendor','Microsoft SQL Server', ...  
    'Server','sname', ...  
    'PortNumber',123456);
```

Create the database metadata object dbmeta using conn.

```
dbmeta = dmd(conn);
```

Retrieve the list of all table names and table types in the catalog and schema using dbmeta. Here, this code assumes that the database contains the catalog name toy_store and schema name sch.

```
catalog = 'toy_store';  
schema = 'sch';  
  
t = tables(dbmeta,catalog,schema)  
  
t =
```

```

'productTable'      'TABLE'
'salesVolume'      'TABLE'
'suppliers'        'TABLE'
...

```

`t` returns a cell array with the table names in the first column and the table types in the second column.

Close the database connection `conn`.

```
close(conn)
```

- “Display Database Metadata” on page 5-38
- “Import Data from Databases into MATLAB” on page 5-3
- “Export Data to New Record in Database” on page 5-21

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with the `database` function.

dbmeta — Database metadata

database metadata object

Database metadata, specified as a database metadata object created using `dmd`. To use this object, create a database connection using the `database` function first.

Note Support for the database metadata object will be removed in a future release. Use the `sqlfind` function and access the properties of the connection object instead.

catalog — Database catalog name

character vector | string scalar

Database catalog name, specified as a character vector or string scalar.

Data Types: `char` | `string`

schema — Database schema name

character vector | string scalar

Database schema name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

t — Table information

cell array

Table information, returned as a cell array with two columns. The first column contains the table names. The second column contains the table types.

See Also

`catalogs` | `close` | `database` | `dmd` | `get` | `schemas` | `sqlfind`

Topics

“Display Database Metadata” on page 5-38

“Import Data from Databases into MATLAB” on page 5-3

“Export Data to New Record in Database” on page 5-21

“Connecting to Database” on page 2-142

“Connecting to Database Using Native ODBC Interface” on page 3-19

Introduced in R2010a

update

Package: database.odbc

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

First, connect to a Microsoft Access database. 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 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

Import all data from the `inventoryTable` using `conn`. Store the data in a cell array contained in the `Data` property of the cursor object. Display the data from `inventoryTable` in this property.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');  
curs = fetch(curs);  
curs.Data
```

```
ans =
```

```
    [ 1]    [1700]    [14.5000]    '2014-09-23 09:38...'  
    [ 2]    [1200]    [      9]    '2014-07-08 22:50...'  
    [ 3]    [ 356]    [     17]    '2014-05-14 07:14...'  
    ...
```

Define a cell array containing the column name that you are updating.

```
colnames = {'Quantity'};
```

Define a cell array containing the new data 2000.

```
data = {2000};
```

Update the column `Quantity` in the `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 the `inventoryTable`.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');  
curs = fetch(curs);  
curs.Data
```

```
ans =
```

```
    [ 1]    [2000]    [14.5000]    '2014-09-23 09:38...'  
    [ 2]    [1200]    [      9]    '2014-07-08 22:50...'
```



```
[ 3]      [ 356]      [      17]      '2014-05-14 07:14...'
...
```

In the `inventoryTable` data, the product with the product number equal to 1 has an updated quantity of 2000 units.

After you finish working with the `cursor` object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

Update Existing Record Using Table

First, connect to a Microsoft Access database. 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 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`

Import all data from the `inventoryTable` using `conn`. Store the data in a cell array contained in the `cursor` object property `Data`. Display the data from `inventoryTable` in this property.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
[ 1] [1700] [14.5000] '2014-09-23 09:38...'  
[ 2] [1200] [      9] '2014-07-08 22:50...'  
[ 3] [ 356] [      17] '2014-05-14 07:14...'  
...
```

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 8:50:15.0'.

```
data = table(15,{'2014-12-01 8:50:15.0'},...  
            '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 the `inventoryTable`.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');  
curs = fetch(curs);  
curs.Data
```

```
ans =
```

```
[ 1] [1700] [      15] '2014-12-01 08:50...'  
[ 2] [1200] [      9] '2014-07-08 22:50...'  
[ 3] [ 356] [      17] '2014-05-14 07:14...'  
...
```

The product with the product number equal to 1 has an updated price of \$15 and timestamp '2014-12-01 8:50:15.0'.

After you finish working with the cursor object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

Update Multiple Records with Multiple Conditions

First, connect to a Microsoft Access database. Store the data that you are updating in a cell array. Then, update multiple records of data in the table using multiple `WHERE` clauses. Close the database connection.

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`

Import all data from the `inventoryTable` using `conn`. Store the data in a cell array contained in the `Data` property of the cursor object. Display the data from `inventoryTable` in this property.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
...
[ 5] [9000] [ 3] '2012-09-14 15:00...'
[ 6] [4540] [ 8] '2013-12-25 19:45...'
[ 7] [6034] [16] '2014-08-06 08:38...'
[ 8] [8350] [ 5] '2011-06-18 11:45...'
...
```

Define a cell array containing the column name that you are updating called `Quantity`.

```
colnames = {'Quantity'};
```

Define a cell array containing the new data. Update 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 the `inventoryTable` for the products with product numbers equal to 5 and 8. Create a cell array `whereclause` that contains two `WHERE` clauses for both products.

```
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`.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
...
[ 5] [10000] [ 3] '2012-09-14 15:00...'
[ 6] [ 4540] [ 8] '2013-12-25 19:45...'
[ 7] [ 6034] [16] '2014-08-06 08:38...'
[ 8] [ 5000] [ 5] '2011-06-18 11:45...'
...
```

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.

After you finish working with the `cursor` object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

Update Multiple Columns with Multiple Conditions

First, connect to a Microsoft Access database. Store the data that you are updating in a cell array. Then, update multiple columns of data in the table using multiple `WHERE` clauses. Close the database connection.

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. This database contains the table `inventoryTable` that contains these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

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

Import all data from `inventoryTable` using `conn`. Store the data in a cell array contained in the `Data` property of the cursor object. Display the data from `inventoryTable` in this property.

```
curs = exec(conn, 'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
...
[ 5] [9000] [ 3] '2012-09-14 15:00...'
[ 6] [4540] [ 8] '2013-12-25 19:45...'
[ 7] [6034] [16] '2014-08-06 08:38...'
[ 8] [8350] [ 5] '2011-06-18 11:45...'
...
```

Define a cell array containing the column names that you are updating called `Quantity` and `Price`.

```
colnames = {'Quantity', 'Price'};
```

Define a cell array containing the new data. Update 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 the `inventoryTable` for the products with product numbers equal to 5 and 8. Create a cell array `whereclause` that contains two `WHERE` clauses for both products.

```
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 the `inventoryTable`.

```
curs = exec(conn,'SELECT * FROM inventoryTable');
curs = fetch(curs);
curs.Data
```

```
ans =
```

```
...
[ 5] [10000] [ 5.5000] '2012-09-14 15:00...'
[ 6] [ 4540] [      8] '2013-12-25 19:45...'
[ 7] [ 6034] [     16] '2014-08-06 08:38...'
[ 8] [ 9000] [     10] '2011-06-18 11:45...'
...
```

The product with the product number equal to 5 has an updated quantity of 10000 units and price equal to 5.50. The product with the product number equal to 8 has an updated quantity of 9000 units and price equal to 10.

After you finish working with the cursor object, close it.

```
close(curs)
```

Close the database connection.

```
close(conn)
```

- “Replace Existing Data in Database” on page 5-25
- “Roll Back Data After Updating Record” on page 5-18
- “Import Data from Databases into MATLAB” on page 5-3

Input Arguments

conn — Database connection

connection object

Database connection, specified as a connection object created with the database function.

tablename — Database table name

character vector | string scalar

Database table name, specified as a character vector or string scalar denoting the name of a table in the database.

Example: 'employees'

Data Types: char | string

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 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
```

See Also

[close](#) | [commit](#) | [database](#) | [datainsert](#) | [get](#) | [rollback](#) | [set](#)

Topics

“Replace Existing Data in Database” on page 5-25

“Roll Back Data After Updating Record” on page 5-18

“Import Data from Databases into MATLAB” on page 5-3

“Connecting to Database Using Native ODBC Interface” on page 3-19

“Data Type Support” on page 1-3

External Websites

[SQL Tutorial](#)

Introduced before R2006a

width

Package: database.odbc

Return field size of column in fetched data set

Syntax

```
colsize = width(curs,colnum)
```

Description

`colsize = width(curs,colnum)` returns the field size of the specified column number `colnum` in the fetched data set `curs`.

Examples

Retrieve the width of the first column of the fetched data set `curs`.

```
colsize = width(curs,1)
```

```
colsize =
```

```
    11
```

The field size of column one is 11 characters (bytes).

To create fetched data sets using an ODBC connection, you can use the native ODBC interface. For details, see `database`.

See Also

`attr` | `cols` | `columnnames` | `fetch` | `get`

Topics

“Display Information About Imported Data” on page 5-50

Introduced before R2006a

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 8-363 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 =
```

```
    1×5 table
```

<u>Catalog</u>	<u>Schema</u>	<u>Table</u>	<u>Columns</u>	<u>Type</u>
'toy_store'	'dbo'	'productTable'	{1×5 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 =
```

```
8×5 table
```

Catalog	Schema	Table	Columns	Type
'toy_store'	'dbo'	'DS17111713025590'	{1×5 cell}	'TABLE'
'toy_store'	'dbo'	'DS17111713025699'	{1×4 cell}	'TABLE'
'toy_store'	'dbo'	'DS22121715025751'	{1×5 cell}	'TABLE'
'toy_store'	'dbo'	'DS22121715025879'	{1×4 cell}	'TABLE'
'toy_store'	'dbo'	'DS22121715052820'	{1×5 cell}	'TABLE'
'toy_store'	'dbo'	'DS22121715052941'	{1×4 cell}	'TABLE'
'toy_store'	'dbo'	'DS26121710493780'	{1×5 cell}	'TABLE'
'toy_store'	'dbo'	'DS26121710493818'	{1×4 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 =
```

```
1×4 cell array
```

```
{'productNumber'} {'Quantity'} {'Price'} {'inventoryDate'}
```

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.

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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

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

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.

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.

Example: 'Schema', 'dbo'

Data Types: char | string

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	

Definitions

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

See Also

`close` | `database` | `sqlinnerjoin` | `sqlread`

Topics

“Retrieve Database Metadata” on page 5-91

“Import Data from Databases into MATLAB” on page 5-3

“Import Data from Database Table Using `sqlread` Function” on page 5-85

Introduced in R2018a

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 pair arguments. For example, 'Keys', 'productNumber' specifies using 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
1	4.0035e+05	1001	14	'Building Blocks'
2	4.0031e+05	1002	9	'Painting Set'
3	4.01e+05	1009	17	'Slinky'

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
```

<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'

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 =
```

```
3x10 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'

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
```

<u>EMPLOYEE_ID</u>	<u>FIRST_NAME</u>	<u>LAST_NAME</u>	<u>EMAIL</u>	<u>PHONE_NUMBER</u>	<u>HIRE_DATE</u>
--------------------	-------------------	------------------	--------------	---------------------	------------------

101	'Neena'	'Kochhar'	'NKOCHHAR'	'515.123.4568'	'2005-09'
102	'Lex'	'De Haan'	'LDEHAAN'	'515.123.4569'	'2001-01'
104	'Bruce'	'Ernst'	'BERNST'	'590.423.4568'	'2007-05'

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

<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'
8	2.1257e+05	1001	5	'Train Set'

`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)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as a `connection` object created with 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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

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 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"]

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: '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

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.

See Also

`close` | `database` | `sqlfind` | `sqlouterjoin` | `sqlread`

Topics

“Join Tables Using Command Line” on page 5-84

“Fetching Data Common Errors” on page 3-4

Introduced in R2018a

sqlouterjoin

Package: database.odbc

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 pair arguments. For example, 'Keys', 'productNumber' specifies using 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
```

<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'

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
```

<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'

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
1	4.0035e+05	1001	14	'Building Blocks'
2	4.0031e+05	1002	9	'Painting Set'
3	4.01e+05	1009	17	'Slinky'

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 =
```

```
 3×5 table
```

<u>DEPARTMENT_ID</u>	<u>DEPARTMENT_ID_1</u>	<u>DEPARTMENT_NAME</u>	<u>DEPT_MANAGER_ID</u>	<u>LOCATI</u>
----------------------	------------------------	------------------------	------------------------	---------------

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 =
```

```
3×5 table
```

DEPARTMENT_ID	DEPARTMENT_ID_1	DEPARTMENT_NAME	DEPT_MANAGER_ID	LOCATION
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
7	3.8912e+05	1007	16	'Engine Kit'
8	2.1257e+05	1001	5	'Train Set'
9	1.2597e+05	1003	13	'Victorian Doll'

`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)`

Input Arguments

conn — Database connection

connection object

Database connection, specified as a `connection` object created with 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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

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 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"]

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: '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 `sqlouterjoin` function to join tables in the MATLAB workspace.

Example: 'Type', 'left'

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.

See Also

`close` | `database` | `sqlfind` | `sqlinnerjoin` | `sqlread`

Topics

“Join Tables Using Command Line” on page 5-84

“Fetching Data Common Errors” on page 3-4

Introduced in R2018a

sqlread

Package: database.odbc

Import data into MATLAB from 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. Executing this function is the equivalent of writing a `SELECT * FROM tablename` SQL statement in ANSI SQL.

`data = sqlread(conn,tablename,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `'Catalog', 'cat'` imports data from a database table stored in the `'cat'` catalog.

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 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 database table `productTable`. The `sqlread` function returns a MATLAB® table that contains the product data.

```
tablename = 'productTable';  
data = sqlread(conn,tablename);
```

Display the first few products.

```
head(data,3)
```

```
ans =
```

```
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)
```

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 =
```

```
3×1 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
```

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'

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.

tablename — Database table name

character vector | string scalar

Database table name, specified as a character vector or string scalar denoting the name of a table in the database.

Example: 'employees'

Data Types: char | string

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

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

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.

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.

Example: 'Schema', 'dbo'

Data Types: `char` | `string`

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`

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.

See Also

`close` | `database` | `fetch` | `select` | `sqlfind` | `sqlinnerjoin` | `sqlouterjoin`

Topics

"Import Data from Database Table Using `sqlread` Function" on page 5-85

"Data Import Approaches and Memory Management" on page 5-46

Introduced in R2018a

sqlwrite

Package: database.odbc

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 =
```

```
 3x5 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 =
```

```
4x5 table
```

<u>productNumber</u>	<u>stockNumber</u>	<u>supplierNumber</u>	<u>unitCost</u>	<u>productDescription</u>
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 =
```

```
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 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
```

<u>productNumber</u>	<u>stockNumber</u>	<u>supplierNumber</u>	<u>unitCost</u>	<u>productDescription</u>
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
```

<u>productNumber</u>	<u>stockNumber</u>	<u>supplierNumber</u>	<u>unitCost</u>	<u>productDescription</u>
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 =
```

```
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)
```

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	Aug
5.101e+05	235	1800	1040	900	750	700	400	350
8.9975e+05	123	1700	823	701	689	621	545	420
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 =
```

```
3x4 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	Aug
4.7082e+05	3100	9400	1540	1500	1350	1190	900	800
5.101e+05	235	1800	1040	900	750	700	400	350
8.9975e+05	123	1700	823	701	689	621	545	420

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	Aug
4.7082e+05	3100	9400	1540	1500	1350	1190	900	800
5.101e+05	235	1800	1040	900	750	700	400	350
8.9975e+05	123	1700	823	701	689	621	545	420
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

<u>productNumber</u>	<u>stockNumber</u>	<u>supplierNumber</u>	<u>unitCost</u>	<u>productDescription</u>
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)
```

Input Arguments

conn — Database connection

connection object

Database connection, specified as a `connection` object created with the `database` function.

tablename — Database table name

character vector | string scalar

Database table name, specified as a character vector or string scalar denoting the name of a table in the database.

Example: `'employees'`

Data Types: `char` | `string`

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 8-0 and “Data Types for New Table” on page 8-0 .

Note The `sqlwrite` function supports only the `table` data type for the data output 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 8-0 .

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 	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
- 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
Logical array	NUMERIC
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	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
string array	Char, Varchar, or other text data type	missing

Data Types: table

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

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

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.

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.

Example: 'Schema', 'dbo'

Data Types: char | string

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

See Also

array2table | cell2table | close | database | sqlread | struct2table

Topics

"Inserting Data Using Command Line" on page 2-148

"Insert Data into Database Table" on page 5-88

"Append Data to Existing Database Table Using Insert Functionality" on page 5-80

"Insert Data into New Database Table Using Insert Functionality" on page 5-82

"Writing Data Common Errors" on page 3-2

Introduced in R2018a

Neo4jConnect

Neo4j database connection

Description

Create a Neo4j database connection using the MATLAB interface to Neo4j. Explore the graph database or perform graph analytics using the MATLAB directed graph.

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.
- 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.

Example: `http://localhost:7474/db/data` where `localhost` is the server, `7474` is the port number, `/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

nodeLabels	All node labels in Neo4j database
relationTypes	All relationship types in Neo4j database
propertyKeys	All property keys in Neo4j database
searchNodeByID	Search for 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
searchGraph	Search for subgraph or entire graph in Neo4j database
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`.

```
neo4jconn.Message
```

```
ans =  
  
    []
```

The blank `Message` property indicates a successful Neo4j® database connection.

Retrieve all node labels using the Neo4j® database connection `neo4jconn`.

```
nlabels = nodeLabels(neo4jconn)
```

```
nlabels =  
  
    1×1 cell array  
  
    {'Person'}
```

The cell array `nlabels` contains a character vector for the one node label in the Neo4j® database.

- “Determine Dependencies of Services in Network”

- “Find Shortest Path Between People in Social Neighborhood”
- “Find Friends of Friends in Social Neighborhood”
- “Explore Graph Database Structure” on page 6-2

See Also

Topics

“Determine Dependencies of Services in Network”

“Find Shortest Path Between People in Social Neighborhood”

“Find Friends of Friends in Social Neighborhood”

“Explore Graph Database Structure” on page 6-2

“Working with MATLAB Interface to Neo4j” on page 6-9

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

Introduced in R2016b

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

nodeDegree	In-degree and out-degree for each associated relationship type for Neo4j database node
nodeRelationTypes	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: [1×1 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'
```

- “Explore Graph Database Structure” on page 6-2

See Also

Topics

“Explore Graph Database Structure” on page 6-2

“Working with MATLAB Interface to Neo4j” on page 6-9

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

Introduced in R2016b

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

RelationType — 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
```

reinfo 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.

```
reinfo.RelationData
```

```
ans =
```

```
struct with no fields.
```

See Also

Topics

“Explore Graph Database Structure” on page 6-2

“Working with MATLAB Interface to Neo4j” on page 6-9

Introduced in R2018a

neo4j

Connect to Neo4j database

The `neo4j` function creates connections to a Neo4j database. For relational database connections, see “Connecting to Database” on page 2-142.

Syntax

```
neo4jconn = neo4j(url,username,password)
```

Description

`neo4jconn = neo4j(url,username,password)` creates a `Neo4jConnect` object using the URL, user name, and password for the Neo4j database. To retrieve graph data from the Neo4j database, use this object.

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 neo4j conn.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

The blank Message property indicates a successful Neo4j® database connection.

Retrieve all node labels using the Neo4j® database connection neo4j conn.

```
nlabels = nodeLabels(neo4jconn)
```

```
nlabels =
```

```
 1×1 cell array
```

```
 {'Person'}
```

The cell array nlabels contains a character vector for the one node label in the Neo4j® database.

- “Determine Dependencies of Services in Network”
- “Find Shortest Path Between People in Social Neighborhood”
- “Find Friends of Friends in Social Neighborhood”
- “Explore Graph Database Structure” on page 6-2

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.

Example: `http://localhost:7474/db/data` where `localhost` is the server, `7474` is the port number, `/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

neo4jconn — Neo4j database connection

Neo4jConnect object

Neo4j database connection, returned as a Neo4jConnect object.

See Also

`neo4j` | `nodeLabels` | `propertyKeys` | `relationTypes`

Topics

“Determine Dependencies of Services in Network”

“Find Shortest Path Between People in Social Neighborhood”

“Find Friends of Friends in Social Neighborhood”

“Explore Graph Database Structure” on page 6-2

“Working with MATLAB Interface to Neo4j” on page 6-9

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

“MATLAB Interface to Neo4j Error Messages” on page 6-28

Introduced in R2016b

nodeLabels

Package: database.neo4j

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 `relationshipTypes`.

Examples

Retrieve Node Labels 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`.

```
neo4jconn.Message
```

```
ans =
```

```
[]
```

The blank `Message` property indicates a successful connection.

Retrieve all node labels using the Neo4j® database connection `neo4jconn`.

```
nlabels = nodeLabels(neo4jconn)
```

```
nlabels =  
    cell  
    'Person'
```

The cell array `nlabels` contains a character vector for the one node label in the Neo4j® database.

- “Explore Graph Database Structure” on page 6-2

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.

See Also

neo4j | propertyKeys | relationTypes

Topics

“Explore Graph Database Structure” on page 6-2

“Working with MATLAB Interface to Neo4j” on page 6-9

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

“MATLAB Interface to Neo4j Error Messages” on page 6-28

Introduced in R2016b

relationTypes

Package: database.neo4j

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 Relationship Types 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`.

```
neo4jconn.Message
```

```
ans =
```

```
[]
```

The blank `Message` property indicates a successful connection.

Retrieve all relationship types using the Neo4j® database connection `neo4jconn`.

```
rtypes = relationTypes(neo4jconn)
```

```
rtypes =
```

```
  cell
```

```
    'knows'
```

The cell array `rtypes` contains a character vector for the one relationship type in the Neo4j® database.

- “Explore Graph Database Structure” on page 6-2

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.

See Also

neo4j | nodeLabels | propertyKeys

Topics

“Explore Graph Database Structure” on page 6-2

“Working with MATLAB Interface to Neo4j” on page 6-9

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

“MATLAB Interface to Neo4j Error Messages” on page 6-28

Introduced in R2016b

propertyKeys

Package: database.neo4j

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 Property Keys 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`.

```
neo4jconn.Message
```

```
ans =
```

```
[]
```

The blank `Message` property indicates a successful connection.

Retrieve all property keys using the Neo4j® database connection `neo4j conn`.

```
propkeys = propertyKeys(neo4j conn)
```

```
propkeys =
```

```
  2×1 cell array
```

```
    'name'  
    'property'
```

The cell array `propkeys` contains a character vector for the one property key in the Neo4j® database.

- “Explore Graph Database Structure” on page 6-2

Input Arguments

neo4j conn — 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.

See Also

neo4j | nodeLabels | relationTypes

Topics

“Explore Graph Database Structure” on page 6-2

“Working with MATLAB Interface to Neo4j” on page 6-9

“MATLAB Interface to Neo4j Error Messages” on page 6-28

Introduced in R2016b

searchNodeByID

Package: database.neo4j

Search for 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`.

```
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: [1×1 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'
```

- “Explore Graph Database Structure” on page 6-2

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.

See Also

`neo4j` | `nodeDegree` | `nodeRelationTypes`

Topics

“Explore Graph Database Structure” on page 6-2

“Working with MATLAB Interface to Neo4j” on page 6-9

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

Introduced in R2016b

searchNode

Package: database.neo4j

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`.

```
neo4jconn.Message
```



```
ans =
    []
```

The blank `Message` property indicates a successful connection.

Search the database for nodes that have node label `Person` using the Neo4j® database connection `neo4jconn`.

```
nlabel = 'Person';
nodeinfo = searchNode(neo4jconn,nlabel)
```

```
nodeinfo =
```

	NodeLabels	NodeData	NodeObject
0	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
1	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
2	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
3	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
4	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
5	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
6	'Person'	[1x1 struct]	[1x1 database.neo4j.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.

```
node = nodeinfo.NodeData(1);
node{1}
```

```
ans =
```

```
struct with fields:  
  name: 'User1'
```

The structure contains one property key and value.

Access the node information using the row name as an index.

```
nodeinfo.NodeData{'0'}
```

```
ans =
```

```
struct with fields:  
  name: 'User1'
```

The structure contains one property key and value.

Find the node degree for the first database node in the table. Specify outgoing relationships.

```
degree = nodeDegree(nodeinfo.NodeObject(1), 'out')
```

```
degree =
```

```
struct with fields:  
  knows: 2
```

There are two outgoing relationships from the first node in the table with relationship type knows.

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 neo4j conn.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

The blank Message property indicates a successful connection.

Search the database for nodes that have node label Person using the Neo4j® database connection neo4j conn. Filter the results further by the property key and value for a specific person named User2.

```
nlabel = 'Person';

nodeinfo = searchNode(neo4jconn,nlabel,'PropertyKey','name', ...
    'PropertyValue','User2')
```

```
nodeinfo =
```

```
Neo4jNode with properties:
    NodeID: 2
    NodeData: [1×1 struct]
    NodeLabels: 'Person'
```

nodeinfo is a Neo4jNode object that contains node information.

Access the node information.

```
nodeinfo.NodeData
```

```
ans =
```

```
struct with fields:  
  name: 'User2'
```

The structure contains a property key and value for `User2`.

Find the node degree of the outgoing relationships.

```
degree = nodeDegree(nodeinfo, 'out')
```

```
degree =
```

```
struct with fields:  
  knows: 1
```

There is one outgoing relationship type `knows` for `User2`.

- “Explore Graph Database Structure” on page 6-2

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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

```
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 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.

See Also

`neo4j` | `nodeDegree` | `nodeRelationTypes` | `searchGraph` | `searchNodeByID` | `searchRelation`

Topics

“Explore Graph Database Structure” on page 6-2

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

“Working with MATLAB Interface to Neo4j” on page 6-9

“MATLAB Interface to Neo4j Error Messages” on page 6-28

Introduced in R2016b

searchRelation

Package: database.neo4j

Search relationships for Neo4j database node

Syntax

```
reinfo = searchRelation(neo4jconn,nodeinfo,direction)
reinfo = searchRelation(neo4jconn,nodeinfo,direction,Name,Value)
```

Description

`reinfo = 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`.

`reinfo = 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 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`.

```
neo4jconn.Message
```

```
ans =
```

```
    []
```

The blank `Message` property indicates a successful connection.

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: [2×3 table]
        Relations: [1×5 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.

```
reinfo.Nodes
```

```
ans =
```

```
2x3 table
```

	NodeLabels	NodeData	NodeObject
1	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
3	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]

Access the table of relationships.

```
reinfo.Relations
```

```
ans =
```

```
1x5 table
```

	StartNodeID	RelationType	EndNodeID	RelationData	RelationObject
3	1	'knows'	3	[1x1 struct]	[1x1 database.neo4j.Neo4jRelationship]

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 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`.

```
neo4jconn.Message
```

```
ans =
```

```
 []
```

The blank Message property indicates a successful connection.

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]

```

reinfo 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.

```
reinfo.Nodes
```

```
ans =
```

```
4x3 table
```

	NodeLabels	NodeData	NodeObject
0	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
1	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
2	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
3	'Person'	[1x1 struct]	[1x1 database.neo4j.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.Neo4jRelationship]
2	2	'knows'	1	[1x1 struct]	[1x1 database.neo4j.Neo4jRelationship]

```
1      0      'knows'      1      [1x1 struct]      [1x1 database.neo
```

- “Explore Graph Database Structure” on page 6-2

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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `relinfo = 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

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> .

Field	Description
Nodes	<p>Table that contains node information for each node in the <code>Relations</code> table. The</p> <ul style="list-style-type: none"> <code>NodeLabels</code> — Character vector that denotes the node label for each match <code>NodeData</code> — Structure array that contains node information such as property <code>NodeObject</code> — <code>Neo4jNode</code> object that represents each matched database n <p>The row names in the table are Neo4j node identifiers of the matched database</p>
Relations	<p>Table that contains relationship information for the nodes in the <code>Nodes</code> table. The</p> <ul style="list-style-type: none"> <code>StartNodeID</code> — Node identifier for the start node for each matched relation <code>RelationType</code> — Character vector that denotes the relationship type for ea <code>EndNodeID</code> — Node identifier for the end node for each matched relationshi <code>RelationData</code> — Structure array that contains property keys associated wi <code>RelationObject</code> — <code>Neo4jRelation</code> object that represents each matched <p>The row names in the table are Neo4j relationship identifiers.</p>

See Also

`Neo4jRelation` | `neo4j` | `searchGraph` | `searchNode` | `searchNodeByID` | `searchRelationByID`

Topics

“Explore Graph Database Structure” on page 6-2

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

“Working with MATLAB Interface to Neo4j” on page 6-9

“MATLAB Interface to Neo4j Error Messages” on page 6-28

Introduced in R2016b

searchGraph

Package: database.neo4j

Search for subgraph or entire graph in Neo4j database

Syntax

```
graphinfo = searchGraph(neo4jconn,criteria)
```

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.

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 =
```

```
7×3 table
```

	NodeLabels	NodeData	NodeObject
0	'Person'	[1×1 struct]	[1×1 database.neo4j.Neo4jNode]
1	'Person'	[1×1 struct]	[1×1 database.neo4j.Neo4jNode]
2	'Person'	[1×1 struct]	[1×1 database.neo4j.Neo4jNode]
3	'Person'	[1×1 struct]	[1×1 database.neo4j.Neo4jNode]
4	'Person'	[1×1 struct]	[1×1 database.neo4j.Neo4jNode]
5	'Person'	[1×1 struct]	[1×1 database.neo4j.Neo4jNode]


```
9      'Person'      [1x1 struct]      [1x1 database.neo4j.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	Relation
1	0	'knows'	1	[1x1 struct]	[1x1 database.neo4j.Neo4jRelationship]
0	0	'knows'	2	[1x1 struct]	[1x1 database.neo4j.Neo4jRelationship]
3	1	'knows'	3	[1x1 struct]	[1x1 database.neo4j.Neo4jRelationship]
2	2	'knows'	1	[1x1 struct]	[1x1 database.neo4j.Neo4jRelationship]
5	3	'knows'	4	[1x1 struct]	[1x1 database.neo4j.Neo4jRelationship]
4	3	'knows'	5	[1x1 struct]	[1x1 database.neo4j.Neo4jRelationship]
6	5	'knows'	4	[1x1 struct]	[1x1 database.neo4j.Neo4jRelationship]
8	5	'knows'	9	[1x1 struct]	[1x1 database.neo4j.Neo4jRelationship]

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);
```

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: [8x5 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.Neo4jNode]
2	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
1	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
3	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
5	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
4	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
9	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]

Access the table of relationships.

graphinfo.Relations

ans =

8x5 table

	StartNodeID	RelationType	EndNodeID	RelationData	Relation
0	0	'knows'	2	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
1	0	'knows'	1	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
2	2	'knows'	1	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
3	1	'knows'	3	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]

4	3	' knows '	5	[1×1 struct]	[1x1 database.neo4j]
5	3	' knows '	4	[1×1 struct]	[1x1 database.neo4j]
6	5	' knows '	4	[1×1 struct]	[1x1 database.neo4j]
8	5	' knows '	9	[1×1 struct]	[1x1 database.neo4j]

Search the graph for all relationship types in the database.

```
allreltypes = relationTypes(neo4jconn);
```

```
graphinfo = searchGraph(neo4jconn,allreltypes);
```

- “Determine Dependencies of Services in Network”
- “Find Shortest Path Between People in Social Neighborhood”
- “Find Friends of Friends in Social Neighborhood”

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</p> <ul style="list-style-type: none"> <code>NodeLabels</code> — Character vector that denotes the node label for each match <code>NodeData</code> — Structure array that contains node information such as property <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</p> <p>If <code>criteria</code> contains node labels, the output is automatically sorted by <code>StartN</code></p>
Relations	<p>Table that contains relationship information for the nodes in the <code>Nodes</code> table. The</p> <ul style="list-style-type: none"> <code>StartNodeID</code> — Node identifier for the start node for each matched relation <code>RelationType</code> — Character vector that denotes the relationship type for ea <code>EndNodeID</code> — Node identifier for the end node for each matched relationshi <code>RelationData</code> — Structure array that contains property keys associated wi <code>RelationObject</code> — <code>Neo4jRelation</code> object that represents each matched <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 R</p>

See Also

`neo4j` | `nodeLabels` | `relationTypes` | `searchNode` | `searchNodeByID` | `searchRelation`

Topics

“Determine Dependencies of Services in Network”

“Find Shortest Path Between People in Social Neighborhood”

“Find Friends of Friends in Social Neighborhood”

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

“Working with MATLAB Interface to Neo4j” on page 6-9

Introduced in R2016b

executeCypher

Package: database.neo4j

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`.

```
neo4jconn.Message
```

```
ans =
```

```
[]
```

The blank `Message` property indicates a successful connection.

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 =
```

```
  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.

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.

See Also

`neo4j` | `searchGraph` | `searchNode` | `searchNodeByID` | `searchRelation`

Topics

“Working with MATLAB Interface to Neo4j” on page 6-9

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

“MATLAB Interface to Neo4j Error Messages” on page 6-28

Introduced in R2016b

nodeRelationTypes

Package: database.neo4j

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`.

```
neo4jconn.Message
```

```
ans =
```

```
[]
```

The blank `Message` property indicates a successful connection.

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 = nodeRelationTypes(node, 'in')
```

```
nodereltypes =
```

```
  cell
```

```
    'knows'
```

`nodereltypes` returns a list of the relationship types.

- “Explore Graph Database Structure” on page 6-2

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.

See Also

`nodeDegree` | `searchNode` | `searchNodeByID`

Topics

“Explore Graph Database Structure” on page 6-2

“Working with MATLAB Interface to Neo4j” on page 6-9

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

Introduced in R2016b

nodeDegree

Package: database.neo4j

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`.

```
neo4jconn.Message
```

```
ans =
```

```
[]
```

The blank `Message` property indicates a successful connection.

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 = nodeDegree(node, 'in')
```

```
degree =
```

```
  struct with fields:
```

```
    knows: 1
```

`degree` returns a structure with the in-degree for each relationship type.

- “Explore Graph Database Structure” on page 6-2

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.

See Also

[nodeRelationTypes](#) | [searchNode](#) | [searchNodeByID](#)

Topics

[“Explore Graph Database Structure” on page 6-2](#)

[“Working with MATLAB Interface to Neo4j” on page 6-9](#)

[“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12](#)

Introduced in R2016b

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” (MATLAB).

`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`.

```
neo4jconn.Message
```

```
ans =
```



```
[]
```

The blank `Message` property indicates a successful connection.

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 = neo4jStruct2Digraph(relinfo)
```

```
G =
```

```
digraph with properties:
```

```
Edges: [2×3 table]
Nodes: [3×3 table]
```

`G` is a `digraph` object that contains two tables for edges and nodes.

Access the table of edges.

```
G.Edges
```

```
ans =
```

EndNodes		RelationType	RelationData
'0'	'1'	'knows'	[1×1 struct]
'2'	'1'	'knows'	[1×1 struct]

Access the table of nodes.

```
G.Nodes
```

```
ans =
```

Name	NodeLabels	NodeData
'0'	'Person'	[1×1 struct]
'1'	'Person'	[1×1 struct]
'2'	'Person'	[1×1 struct]

Find the shortest path between all nodes in G.

```
d = distances(G)
```

```
d =
```

0	1	Inf
Inf	0	Inf
Inf	1	0

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`.

```
neo4jconn.Message
```

```
ans =
```

```
[]
```

The blank Message property indicates a successful connection.

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 = neo4jStruct2Digraph(graphinfo)
```

```
G =
```

```
digraph with properties:
```

```
Edges: [8×3 table]
```

```
Nodes: [7×3 table]
```

`G` is a `digraph` object that contains two tables for edges and nodes.

Access the table of edges.

```
G.Edges
```

```
ans =
```

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'	'6'	'knows'	[1×1 struct]

Access the table of nodes.

G. Nodes

```
ans =
```

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]
'6'	'Person'	[1x1 struct]

Find the shortest path between all nodes in G.

```
d = distances(G)
```

```
d =
```

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

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 `neo4j conn`.

```
neo4j conn.Message
```

```
ans =
```

```
    []
```

The blank `Message` property indicates a successful connection.

Search for a subgraph using the Neo4j® database connection `neo4j conn` and node label `nlabel`.

```
nlabel = {'Person'};
```

```
graphinfo = searchGraph(neo4j conn,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`.

```
names = [graphinfo.Nodes.NodeData{:}];
```

```
nodenames = {names(:).name};
```

```
G = neo4jStruct2Digraph(graphinfo,'NodeNames',nodenames)
```

```
G =
```

```
    digraph with properties:
```

```
    Edges: [8×3 table]
```

```
    Nodes: [7×3 table]
```

`G` is a `digraph` object that contains two tables for edges and nodes.

Access the table of edges.

```
G.Edges
```

```
ans =
```

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'	7

Access the table of nodes.

G.Nodes

ans =

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 =

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

- “Determine Dependencies of Services in Network”
- “Find Shortest Path Between People in Social Neighborhood”
- “Find Friends of Friends in Social Neighborhood”

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.

See Also

`distances` | `neo4j` | `searchGraph` | `searchNode` | `searchRelation`

Topics

“Determine Dependencies of Services in Network”

“Find Shortest Path Between People in Social Neighborhood”

“Find Friends of Friends in Social Neighborhood”

“Searching Graph Database Using MATLAB Interface to Neo4j” on page 6-12

“Working with MATLAB Interface to Neo4j” on page 6-9

Introduced in R2016b

createNode

Package: database.neo4j

Create nodes in Neo4j database

Syntax

```
nodeinfo = createNode(neo4jconn)
nodeinfo = createNode(neo4jconn,Name,Value)
```

Description

`nodeinfo = createNode(neo4jconn)` creates a single node without labels and properties by using a Neo4j database connection.

`nodeinfo = createNode(neo4jconn,Name,Value)` creates a single node or multiple nodes by specifying additional options using one or more name-value pair arguments. For example, 'Labels', 'Person' creates a node with the Person node label.

Examples

Create Node in Neo4j Database

Create a single node in the 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 `neo4j conn`. The blank `Message` property indicates a successful connection.

```
neo4j conn.Message
```

```
ans =  
  
    []
```

Create a single node in the Neo4j database using the Neo4j database connection.

```
nodeinfo = createNode(neo4j conn)
```

```
nodeinfo =  
  
    Neo4jNode with properties:  
  
        NodeID: 30  
        NodeData: [1×1 struct]  
        NodeLabels: []
```

`nodeinfo` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

When you execute the `createNode` function without any arguments except the Neo4j database connection, the function creates a single node without labels and properties. To add labels and properties, use the `updateNode` function.

Create Node with Label in Neo4j Database

Create a single node with a node 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 neo4j conn. The blank Message property indicates a successful connection.

```
neo4jconn.Message
```

```
ans =

    []
```

Create a single node in the Neo4j database with a node label using the Neo4j database connection. Use the 'Labels' name-value pair argument to specify the Person node label.

```
label = 'Person';
nodeinfo = createNode(neo4jconn,'Labels',label)
```

```
nodeinfo =

    Neo4jNode with properties:
        NodeID: 31
        NodeData: [1x1 struct]
        NodeLabels: 'Person'
```

nodeinfo is a Neo4jNode object with these properties:

- Node identifier
- Node data
- Node labels

Display the node label for this node.

```
nodeinfo.NodeLabels
```

```
ans =  
    'Person'
```

Create Two Nodes with Labels in Neo4j Database

Create two nodes with labels in the 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 Neo4j database by using the Neo4j database connection. Use the `'Labels'` name-value pair argument to specify the node labels. One node has the `Person` node label and the other node has two node labels, `Person` and `Employee`.

```
labels = {'Person'},{'Person','Employee'};  
nodeinfo = createNode(neo4jconn,'Labels',labels)
```

```
nodeinfo =  
    2×3 table
```

	NodeLabels	NodeData	NodeObject
32	{1×1 cell}	[1×1 struct]	[1×1 database.neo4j.Neo4jNode]
33	{2×1 cell}	[1×1 struct]	[1×1 database.neo4j.Neo4jNode]

`nodeinfo` is a table with two rows, one for each person. The table contains these variables:

- Node labels
- Node properties
- `Neo4jNode` object

Access the `Neo4jNode` object for the first node.

```
data = nodeinfo.NodeObject(1)
```

```
data =
```

```
Neo4jNode with properties:
```

```
NodeID: 32
NodeData: [1×1 struct]
NodeLabels: {'Person'}
```

`data` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

Access the node labels of both nodes.

```
nodeinfo.NodeObject(1).NodeLabels
nodeinfo.NodeObject(2).NodeLabels
```

```
ans =
```

```
1×1 cell array
```

```
    {'Person'}

ans =

    2×1 cell array

    {'Person' }
    {'Employee'}
```

Create Two Nodes with Labels and Properties in Neo4j Database

Create two nodes with labels and properties in the 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 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 =

    []
```

Create a table with two rows that contain the names and job titles of two people.

```
props = table(["User8";"User9"],["Analyst";"Technician"], ...
    'VariableNames',{ 'Name', 'Title'});
```

Create two nodes that represent these two people in the Neo4j 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
8	{2x1 cell}	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
34	{2x1 cell}	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]

nodeinfo is a table with two rows, one for each person. The table contains these variables:

- Node labels
- Node properties
- 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: 'Technician'
Name: 'User9'
```

Access the `Neo4jNode` object for the first node.

```
data = nodeinfo.NodeObject(1)
```

```
data =
```

```
Neo4jNode with properties:
```

```
    NodeID: 8  
    NodeData: [1×1 struct]  
    NodeLabels: {2×1 cell}
```

`data` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `node =`

```
createNode(neo4jconn, 'Labels', 'Person', 'Properties', table(["User8"],  
["Analyst"], 'VariableNames', {'Name', 'Title'})) creates a single node with  
the Person node label and two properties, Name and Title, and 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 string array. To create multiple nodes with different node labels, use a cell array of cell arrays or 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, each field and its corresponding value convert to a property and its corresponding value in the database node. Also, the `createNode` function sets the `NodeData` property of the `Neo4jNode` object to this structure.

When you specify a table that contains one row, each variable and its corresponding value convert to a property and its corresponding value in the database node. The `createNode` function 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 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.

See Also

`deleteNode` | `neo4j` | `updateNode`

Topics

“Update Friend Information in Social Neighborhood” on page 6-15

“Working with MATLAB Interface to Neo4j” on page 6-9

Introduced in R2018a

createRelation

Package: database.neo4j

Create relationships between nodes in Neo4j database

Syntax

```
relationinfo = createRelation(neo4jconn,startnode,endnode,  
relationtype)  
relationinfo = createRelation(neo4jconn,startnode,endnode,  
relationtype,'Properties',properties)
```

Description

`relationinfo = createRelation(neo4jconn,startnode,endnode,relationtype)` creates a single relationship or multiple relationships between the start and end nodes with specified relationship types using the Neo4j database connection.

`relationinfo = createRelation(neo4jconn,startnode,endnode,relationtype,'Properties',properties)` specifies the properties of the new relationships.

Examples

Create Relationship Between Two Nodes in Neo4j Database

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 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. Specify the relationship type as `works with`.

```
relationtype = 'works with';  
relationinfo = createRelation(neo4jconn,startnode,endnode,relationtype)
```

```
relationinfo =  
  
Neo4jRelation with properties:  
  
RelationID: 31  
RelationData: [1x1 struct]  
StartNodeID: 39  
RelationType: 'works with'  
EndNodeID: 40
```

`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 =
    'works with'
```

Create Multiple Relationships in Neo4j Database

Create two relationships between nodes in a Neo4j® database 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 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 for the node with the node 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 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';
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 =`

2×5 table

	StartNodeID	RelationType	EndNodeID	RelationData	RelationInfo
33	41	'studies with'	42	[1×1 struct]	[1×1 database.r
32	41	'works with'	9	[1×1 struct]	[1×1 database.r

`relationinfo` is a table with these variables:

- Start node identifier
- Relationship type
- End node identifier

- Relationship properties
- Neo4jRelation object

Display the Neo4jRelation object for the first relationship.

```
relation = relationinfo.RelationObject(1)
```

```
relation =
```

```
Neo4jRelation with properties:
```

```
RelationID: 33  
RelationData: [1x1 struct]  
StartNodeID: 41  
RelationType: 'studies with'  
EndNodeID: 42
```

relation is a Neo4jRelation object with these properties:

- Relationship identifier
- Relationship data
- Start node identifier
- Relationship type
- End node identifier

Create Relationship with Properties in Neo4j Database

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 `neo4j conn`. The blank `Message` property indicates a successful connection.

```
neo4j conn.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(neo4j conn, 'Labels', label);  
endnode = createNode(neo4j conn, 'Labels', label);
```

Create a relationship between the two nodes using the Neo4j database connection. These nodes represent two colleagues that 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';  
relationinfo = createRelation(neo4j conn, startnode, endnode, relationtype, ...  
    'Properties', properties)
```

```
relationinfo =
```

```
    Neo4jRelation with properties:
```

```
        RelationID: 34  
        RelationData: [1x1 struct]  
        StartNodeID: 43  
        RelationType: 'works with'  
        EndNodeID: 44
```

`relationinfo` is a `Neo4jRelation` object with these properties:

- Relationship identifier
- Relationship data
- Start node identifier
- Relationship type
- End node identifier

Display the properties of the relationship.

```
relationinfo.RelationData
```

```
ans =
```

```
  struct with fields:
```

```
    StartDate: '09/01/2017'
    Project: 'Database'
```

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 this object for one node or an array of these 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 that the createRelation function creates 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 this object for one node or an array of these 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 that the `createRelation` function creates in the Neo4j database.

Example: 9

relationshiptype — 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 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, each field and its corresponding value convert to a property and its corresponding value in the database relationship.

When you specify a table with one row, each variable and its corresponding value convert to a property and its corresponding value in the database relationship.

The `createRelation` function 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 resulting relationship does not contain this property.

Data Types: `struct` | `table` | `cell`

Output Arguments

relationinfo — Relationship

`Neo4jRelation` object | table

Relationship, returned as a `Neo4jRelation` object or table that contains relationship information. A `Neo4jRelation` object represents a single relationship. A table represents multiple relationships and contains these variables:

- `StartNodeID` — Node identifier for the start node for each matched relationship
- `RelationType` — Character vector that denotes the relationship type for each matched relationship
- `EndNodeID` — Node identifier for 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.

See Also

`deleteRelation` | `neo4j` | `updateRelation`

Topics

“Update Friend Information in Social Neighborhood” on page 6-15

“Working with MATLAB Interface to Neo4j” on page 6-9

Introduced in R2018a

deleteNode

Package: database.neo4j

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 in Neo4j Database

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 `neo4j conn`. The blank `Message` property indicates a successful connection.

```
neo4j conn.Message
```

```
ans =
```

```
    []
```

Create a single node in the Neo4j database using the Neo4j database connection.

```
node = createNode(neo4j conn)
```

```
node =
```

```
    Neo4jNode with properties:
```

```
        NodeID: 35
```

```
        NodeData: [1x1 struct]
```

```
        NodeLabels: []
```

`node` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

Delete the node using the Neo4j database connection.

```
deleteNode(neo4j conn, node)
```

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: 23
    RelationData: [1x1 struct]
    StartNodeID: 36
    RelationType: 'works with'
    EndNodeID: 37
```

`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)
```

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 numeric vector. For one node, specify a Neo4jNode object or numeric scalar. For multiple nodes, specify a Neo4jNode object array or numeric vector.

The numeric scalar or vector must contain Neo4j database node identifiers.

Example: 15

Example: [2, 3, 4]

See Also

createNode | neo4j | updateNode

Topics

“Update Friend Information in Social Neighborhood” on page 6-15

“Working with MATLAB Interface to Neo4j” on page 6-9

Introduced in R2018a

deleteRelation

Package: database.neo4j

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: 24  
        RelationData: [1×1 struct]  
        StartNodeID: 21  
        RelationType: 'works with'  
        EndNodeID: 22
```

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)
```

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 numeric scalar that contains the relationship identifier. For multiple relationships, use a `Neo4jRelation` object array or numeric vector that contains an array of relationship identifiers.

Example: 15

Example: [15,16,17]

See Also

`createRelation` | `neo4j` | `updateRelation`

Topics

“Update Friend Information in Social Neighborhood” on page 6-15

“Working with MATLAB Interface to Neo4j” on page 6-9

Introduced in R2018a

updateNode

Package: database.neo4j

Update node labels and properties in Neo4j database

Syntax

```
nodeinfo = updateNode(neo4jconn,node,'Labels',labels)
nodeinfo = updateNode(neo4jconn,node,'Properties',properties)
nodeinfo = updateNode(neo4jconn,node,'Labels',labels,'Properties',
properties)
```

Description

`nodeinfo = updateNode(neo4jconn,node,'Labels',labels)` updates existing node labels by replacing them with the specified node labels for associated nodes by using a Neo4j database connection.

`nodeinfo = updateNode(neo4jconn,node,'Properties',properties)` updates existing node properties by replacing them with the specified node properties.

`nodeinfo = updateNode(neo4jconn,node,'Labels',labels,'Properties',properties)` updates existing node labels and properties.

Examples

Update Node Labels in Neo4j Database

Create a single node in a Neo4j® database, update its node labels, and display them.

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: 16  
    NodeData: [1×1 struct]  
    NodeLabels: []
```

`node` is a `Neo4jNode` object with these properties:

- Node identifier
- Node data
- Node labels

Update the node with the labels `Person` and `Employee`. The output argument `nodeinfo` is a `Neo4jNode` object.

```
labels = ["Person","Employee"];  
nodeinfo = updateNode(neo4jconn,node,'Labels',labels);
```

Display the updated node labels.

```
nodeinfo.NodeLabels
```

```
ans =
    2x1 cell array
    {'Person' }
    {'Employee'}
```

Update Node Properties in Neo4j Database

Create a single node in a Neo4j® database, update its 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 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 =
    []
```

Create a single node in the Neo4j database using the Neo4j database connection.

```
node = createNode(neo4jconn)
```

```
node =
```

```
Neo4jNode with properties:
```

```
    NodeID: 23
   NodeData: [1×1 struct]
  NodeLabels: []
```

node is a Neo4jNode object with these properties:

- Node identifier
- Node data
- Node labels

Update the properties of the 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'
```

Update Node 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 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 for a single node with the node label `Person`. Then, using the Neo4j database connection, filter the results further by the property key and value for a specific 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. The `nodeinfo` output argument is a `Neo4jNode` object.

```
properties.title = 'Analyst';  
nodeinfo = updateNode(neo4jconn,node,'Properties',properties);
```

Display the updated node properties.

```
nodeinfo.NodeData
```

```
ans =  
  struct with fields:  
    name: 'User7'  
    title: 'Analyst'
```

Update Node Labels and Properties in Neo4j Database

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 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 =  
  
    []
```

Create a single node in the Neo4j database using the Neo4j database connection.

```
node = createNode(neo4jconn)
```

```
node =  
  
    Neo4jNode with properties:  
  
        NodeID: 55  
        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 with the labels Person and Employee and with the node properties defined in the table. The nodeinfo output argument is a Neo4jNode object.

```
labels = ["Person", "Employee"];  
nodeinfo = updateNode(neo4jconn, node, 'Labels', labels, ...  
    'Properties', properties);
```

Display the updated node labels.

```
nodeinfo.NodeLabels
```

```
ans =
```

```
2×1 cell array
```

```
{'Person' }  
{'Employee'}
```

Display the updated node properties.

```
nodeinfo.NodeData
```

```
ans =
```

```
struct with fields:
```

```
Title: 'Analyst'  
Name: 'User8'
```

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 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 =
```

```
 []
```

Create two nodes in the Neo4j 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 with the labels Person and Employee and with the node properties defined in the table.

```
labels = ["Person","Employee"];
nodeinfo = updateNode(neo4jconn,[user8;user9],'Labels',labels, ...
    'Properties',properties)
```

```
nodeinfo =
```

2×3 table

	NodeLabels	NodeData	NodeObject
56	{2×1 cell}	[1×1 struct]	[1×1 database.neo4j.Neo4jNode]
57	{2×1 cell}	[1×1 struct]	[1×1 database.neo4j.Neo4jNode]

`nodeinfo` is a table that contains these variables:

- Node labels
- Node properties
- `Neo4jNode` objects

Display the node labels for the first node.

```
nodeinfo.NodeLabels{1}
```

```
ans =
```

```
2×1 cell array
```

```
 {'Person' }  
 {'Employee'}
```

Display the node properties for the first node.

```
nodeinfo.NodeData{1}
```

```
ans =
```

```
struct with fields:
```

```
 Title: 'Analyst'
```

Name: 'User8'

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 numeric vector. For one node, specify a Neo4jNode object or numeric scalar. For multiple nodes, specify a Neo4jNode object array or 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 string array. To update multiple nodes with different node labels, use a cell array of cell arrays or 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, each field and its corresponding value convert to a property and its corresponding value in the database node. The `updateNode` function sets the `NodeData` property of the `Neo4jNode` object to this structure.

When you specify a table that contains one row, each variable and its corresponding value convert to a property and its corresponding value in the database node. The `updateNode` function 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 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.

See Also

`createNode` | `deleteNode` | `neo4j`

Topics

“Update Friend Information in Social Neighborhood” on page 6-15

“Working with MATLAB Interface to Neo4j” on page 6-9

Introduced in R2018a

updateRelation

Package: database.neo4j

Update relationship properties in Neo4j database

Syntax

```
relationinfo = updateRelation(neo4jconn,relation,properties)
```

Description

`relationinfo = updateRelation(neo4jconn,relation,properties)` updates the properties of a single relationship or multiple relationships using a Neo4j database connection.

Examples

Update Relationship Between Two Nodes in Neo4j Database

Create a single relationship between two nodes in a Neo4j® database, update the properties of the relationship, 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 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. Specify the relationship type as works with. The relationship 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: 25
 RelationData: [1x1 struct]
 StartNodeID: 26
 RelationType: 'works with'
 EndNodeID: 27
```

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'
```

Update Multiple Relationships in Neo4j Database

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 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 for the node with the node 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 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';  
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`

The relationship `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 =
```

```
2×5 table
```

	<u>StartNodeID</u>	<u>RelationType</u>	<u>EndNodeID</u>	<u>RelationData</u>	<u>Relation</u>
30	28	'studies with'	38	[1×1 struct]	[1×1 database.r
29	28	'works with'	9	[1×1 struct]	[1×1 database.r

relationinfo is a table with these variables:

- Start node identifier
- Relationship type
- End node identifier
- Relationship properties
- Neo4jRelation object

Display the updated properties of the relationships.

```
relationinfo.RelationData{1}  
relationinfo.RelationData{2}
```

```
ans =
```

```
struct with fields:
```

```
    Date: '01/02/2017'  
    Location: 'Workplace'
```

```
ans =
```

```
struct with fields:
```

```
    Date: '03/06/2017'
```

Location: 'Library'

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 numeric scalar that contains the relationship identifier. For multiple relationships, use a Neo4jRelation object array or 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, each field and its corresponding value convert to a property and its corresponding value in the database relationship.

When you specify a table with one row, each variable and its corresponding value convert to a property and its corresponding value in the database relationship.

The updateRelation function 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 resulting relationship does not contain this property.

Data Types: `struct` | `table` | `cell`

Output Arguments

relationinfo — Relationship

`Neo4jRelation` object | `table`

Relationship, returned as a `Neo4jRelation` object or table that contains relationship information. A `Neo4jRelation` object represents a single relationship. A table represents multiple relationships and contains these variables:

- `StartNodeID` — Node identifier for the start node for each matched relationship
- `RelationType` — Character vector that denotes the relationship type for each matched relationship
- `EndNodeID` — Node identifier for 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.

See Also

`createRelation` | `deleteRelation` | `neo4j`

Topics

“Update Friend Information in Social Neighborhood” on page 6-15

“Working with MATLAB Interface to Neo4j” on page 6-9

Introduced in R2018a

searchRelationByID

Package: database.neo4j

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 in Neo4j Database

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 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 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	Relation
5	3	' knows '	4	[1x1 struct]	[1x1 database.neo4j]
4	3	' knows '	5	[1x1 struct]	[1x1 database.neo4j]
6	5	' knows '	4	[1x1 struct]	[1x1 database.neo4j]

relationinfo is a table with these variables:

- Start node identifier
- Relationship type
- End node identifier
- Relationship properties
- Neo4jRelation object

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

Neo4jRelation object | table

Relationship, returned as a `Neo4jRelation` object or table that contains relationship information. A `Neo4jRelation` object represents a single relationship. A table represents multiple relationships and contains these variables:

- `StartNodeID` — Node identifier for the start node for each matched relationship
- `RelationType` — Character vector that denotes the relationship type for each matched relationship
- `EndNodeID` — Node identifier for 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.

See Also

`neo4j` | `searchNodeByID` | `searchRelation`

Topics

“Explore Graph Database Structure” on page 6-2

“Working with MATLAB Interface to Neo4j” on page 6-9

Introduced in R2018a

storeDigraph

Package: database.neo4j

Store directed graph in Neo4j database

Syntax

```
graphinfo = storeDigraph(neo4jconn,G)
graphinfo = storeDigraph(neo4jconn,G,Name,Value)
```

Description

`graphinfo = storeDigraph(neo4jconn,G)` converts a directed graph to a Neo4j graph and stores it in the 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.

`graphinfo = 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.

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 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 =
```

```
    []
```

Create a `digraph` object with three nodes, which represents a new Neo4j graph. These 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 in the `digraph` object as a Neo4j graph in the Neo4j database.

```
graphinfo = storeDigraph(neo4jconn,G)
```

```
graphinfo =
```

```
    struct with fields:
```

```
        Nodes: [3×3 table]
        Relations: [3×5 table]
```

`graphinfo` is a structure that contains node and relationship information. By default, the `storeDigraph` function stores the directed graph without node labels. Also, the function stores the relationships with the default relationship type `Edge` in the resulting Neo4j graph.

Display information about the Neo4j graph nodes.

```
graphinfo.Nodes
```

```
ans =
```

```
3x3 table
```

	<u>NodeLabels</u>	<u>NodeData</u>	<u>NodeObject</u>
45	[]	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
46	[]	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
47	[]	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]

Nodes is a table that contains these variables:

- Node labels
- Node properties
- Neo4jNode objects

Display information about the Neo4j graph relationships.

```
graphinfo.Relations
```

```
ans =
```

```
3x5 table
```

	<u>StartNodeID</u>	<u>RelationType</u>	<u>EndNodeID</u>	<u>RelationData</u>	<u>RelationObject</u>
35	45	'Edge'	46	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
17	45	'Edge'	47	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
18	47	'Edge'	46	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]

Relations is a table that contains these variables:

- Start node identifiers
- Relationship types
- End node identifiers

- Relationship properties
- Neo4jRelation objects

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 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 =
```

```
 []
```

Create a `digraph` object with three nodes, which represents a new Neo4j graph. These 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 in the `digraph` object 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 'GlobalRelationType' name-value pair argument.

```
graphinfo = storeDigraph(neo4jconn,G,'GlobalNodeLabel','Person', ...
    'GlobalRelationType','knows')
```

```
graphinfo =
```

```
    struct with fields:
```

```
        Nodes: [3x3 table]
        Relations: [3x5 table]
```

Display information about the Neo4j graph nodes.

```
graphinfo.Nodes
```

```
ans =
```

```
    3x3 table
```

	NodeLabels	NodeData	NodeObject
48	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
49	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
50	'Person'	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]

Nodes is a table that contains these variables:

- Node labels
- Node properties
- Neo4jNode objects

Display information about the Neo4j graph relationships.

```
graphinfo.Relations
```

```
ans =
```

3×5 table

	StartNodeID	RelationType	EndNodeID	RelationData	Relation
	37	48	'knows'	50	[1×1 struct] [1×1 database.ne
	36	48	'knows'	49	[1×1 struct] [1×1 database.ne
	19	50	'knows'	49	[1×1 struct] [1×1 database.ne

Relations is a table that contains these variables:

- Start node identifiers
- Relationship types
- End node identifiers
- Relationship properties
- Neo4jRelation objects

Store Directed Graph with Node Labels and Relationship Types

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.

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 =
```

```
    []
```

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 node and edge 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
51	{2x1 cell}	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
52	{2x1 cell}	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]
53	{2x1 cell}	[1x1 struct]	[1x1 database.neo4j.Neo4jNode]

Nodes is a table that contains these variables:

- Node labels
- Node properties
- Neo4jNode objects

Display information about the Neo4j graph relationships.

```
graphinfo.Relations
```

```
ans =
```

```
2x5 table
```

	StartNodeID	RelationType	EndNodeID	RelationData	RelationObject
26	51	'knows'	53	[1x1 struct]	[1x1 database.neo4j.Neo4jRelation]
27	52	'works with'	53	[1x1 struct]	[1x1 database.neo4j.Neo4jRelation]

Relations is a table that contains these variables:

- Start node identifiers
- Relationship types
- End node identifiers
- Relationship properties
- Neo4jRelation objects

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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

Example: graphinfo = storeDigraph(neo4jconn,G, 'GlobalNodeLabel', 'Person', 'GlobalRelationship', '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 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 string array.

To specify different node 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 type of 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 <code>Relations</code> table. The</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 <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 <code>Nodes</code> table. The</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 <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.

See Also

`createNode` | `createRelation` | `deleteNode` | `deleteRelation` | `neo4j` | `updateNode` | `updateRelation`

Topics

"Add and Query Group of Colleagues in Social Neighborhood" on page 6-20

"Working with MATLAB Interface to Neo4j" on page 6-9

Introduced in R2018a

mongo

MongoDB connection

Description

The `mongo` function creates a `mongo` object using the Database Toolbox interface for MongoDB. With the object, you can connect to MongoDB stored on one or more database servers.

First, you must install the Database Toolbox interface for MongoDB. For details, see “Database Toolbox Interface for MongoDB Installation” on page 7-19.

Using the `mongo` object, you can manage 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

Syntax

```
conn = mongo(server,port,dbname)
conn = mongo(server,port,dbname,'UserName',username,'Password',
password)
```

Description

`conn = mongo(server,port,dbname)` creates a MongoDB connection to the database server using a database name and sets the Port property.

`conn = mongo(server,port,dbname,'UserName',username,'Password',password)` specifies a user name and password for connecting to MongoDB.

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

dbname — Database name

string scalar

Database name, specified as a string scalar.

Example: "employeesdb"

Data Types: string

username — User name

string scalar

User name, specified as a string scalar. Contact your MongoDB administrator for access credentials.

Example: "username"

Data Types: string

password — Password

string scalar

Password, specified as a string scalar. Contact your MongoDB administrator for access credentials.

Example: "pwd"

Data Types: string

Properties

Database — Database name

character vector

This property is read-only.

Database name, specified as a character vector.

The `dbname` input argument sets this property.

Example: `'databasename'`

Data Types: `char`

UserName — User name

character vector

This property is read-only.

User name, specified as a character vector.

The `username` input argument sets this property.

Example: `'username'`

Data Types: `char`

Server — Server name

cell array of character vectors

This property is read-only.

Server name, specified as a cell array of character vectors. Each character vector in the cell array specifies one database server name.

The `server` input argument sets this property.

Example: `{'server1'}`

Data Types: `cell`

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.

Example: 27017

Data Types: double

CollectionNames — Collection names

cell array of character vectors

This property is read-only.

Collection names of all collections defined in MongoDB, specified as a cell array of character vectors.

Example: {'airlinesmall', 'employee', 'largedata' ... and 3 more}

Data Types: cell

TotalDocuments — Count of documents in all collections

numeric scalar

This property is read-only.

Count of the documents in all collections defined in MongoDB, specified as a numeric scalar.

Data Types: double

Object Functions

MongoDB Connection

isopen Determine if MongoDB connection is open

close Close MongoDB connection

Import Document Collections into MATLAB

count Count total number of documents in MongoDB collection

distinct Retrieve distinct values for field in MongoDB collection

find Retrieve documents in MongoDB collection

Export and Manage Document Collections in MongoDB

<code>createCollection</code>	Create MongoDB collection
<code>dropCollection</code>	Drop MongoDB collection
<code>insert</code>	Insert one or multiple documents into MongoDB collection
<code>remove</code>	Remove one or multiple documents from MongoDB collection
<code>update</code>	Update one or multiple documents in MongoDB collection

Examples

Create MongoDB Connection

Connect to MongoDB and count the total number of documents in a collection.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
      Database: 'mongotest'
      UserName: ''
      Server: {'dbtb01'}
      Port: 27017
      CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
      TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.

- This database contains 23,485,919 documents.

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 `employee` collection. There are 25 documents in the collection.

```
collection = "employee";  
n = count(conn, collection)
```

```
n =
```

```
    25
```

Close the MongoDB connection.

```
close(conn)
```

Create MongoDB Connection Using User Name and Password

Connect to MongoDB 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`. Here, the database server `dbtb01` hosts this database using port number 27017. Specify the user name `adminuser` and password `matlab`.

```
conn = mongo("dbtb01", 27017, "mongotest", 'UserName', "adminuser", 'Password', "matlab")
```

```
conn =
```

```
    mongo with properties:
```

```
        Database: 'mongotest'
```

```
UserName: 'adminuser'  
Server: {'dbtb01'}  
Port: 27017  
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
TotalDocuments: 23485919
```

`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 `adminuser`.
- The database server is `dbtb01`.
- The port number is `27017`.
- This database contains six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

Check 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 `employee` collection. There are 25 documents in the collection.

```
collection = "employee";  
n = count(conn, collection)
```

```
n =
```

```
25
```

Close the MongoDB connection.

`close(conn)`

See Also

Topics

“Database Toolbox Interface for MongoDB Installation” on page 7-19

“Import and Analyze Data from MongoDB” on page 7-2

“Import Filtered Data from MongoDB” on page 7-5

“Import Large Data from MongoDB” on page 7-8

“Export MATLAB Data into MongoDB” on page 7-11

“Import and Export MATLAB Objects Using MongoDB” on page 7-15

“Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

MongoDB Manual

Introduced in R2017b

close

Close MongoDB connection

Syntax

```
close(conn)
```

Description

`close(conn)` closes the MongoDB connection.

Examples

Close MongoDB Connection

Connect to MongoDB and count the total number of documents in a collection.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
    Database: 'mongotest'  
    UserName: ''  
    Server: {'dbtb01'}  
    Port: 27017  
    CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
    TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection. There are 25 documents in the collection.

```
collection = "employee";  
n = count(conn, collection)
```

```
n =
```

```
    25
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB connection

mongo object

MongoDB connection, specified as a mongo object.

See Also

[find](#) | [isopen](#) | [mongo](#)

Topics

[“Import and Analyze Data from MongoDB” on page 7-2](#)

[“Import Filtered Data from MongoDB” on page 7-5](#)

[“Import Large Data from MongoDB” on page 7-8](#)

[“Database Toolbox Interface for MongoDB Error Messages” on page 7-21](#)

External Websites

[MongoDB Manual](#)

Introduced in R2017b

isopen

Determine if MongoDB 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 Connection

Connect to MongoDB and count the total number of documents in a collection.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
      Database: 'mongotest'  
      Username: ''  
      Server: {'dbtb01'}  
      Port: 27017  
      CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
      TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection. There are 25 documents in the collection.

```
collection = "employee";  
n = count(conn, collection)
```

```
n =
```

```
25
```

Close the MongoDB connection.

```
close(conn)
```

Verify the MongoDB connection is closed.

```
isopen(conn)
```

```
ans =
```

```
logical
```

0

Input Arguments

conn — MongoDB connection

mongo object

MongoDB connection, specified as a mongo object.

See Also

`close` | `find` | `mongo`

Topics

“Import and Analyze Data from MongoDB” on page 7-2

“Import Filtered Data from MongoDB” on page 7-5

“Import Large Data from MongoDB” on page 7-8

“Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

MongoDB Manual

Introduced in R2017b

createCollection

Create MongoDB collection

Syntax

```
createCollection(conn,collection)
```

Description

`createCollection(conn,collection)` creates a collection in MongoDB by using the MongoDB connection.

Examples

Create Collection in MongoDB

Connect to MongoDB and create a collection.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
    Database: 'mongotest'  
    UserName: ''  
    Server: {'dbtb01'}  
    Port: 27017  
    CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
    TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 =
```

```
1×7 cell array
```

```
Columns 1 through 5
```

```
 {'airlinesmall'}  {'employee'}  {'largedata'}  {'nyctaxi'}  {'product'}
```

```
Columns 6 through 7
```

```
 {'restaurants'}  {'taxidata'}
```


Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB connection

mongo object

MongoDB connection, specified as a mongo object.

collection — Collection name

string scalar

Collection name, specified as a string scalar.

Example: "taxidata"

Data Types: string

See Also

`close` | `dropCollection` | `find` | `insert` | `isopen` | `mongo`

Topics

"Import and Analyze Data from MongoDB" on page 7-2

"Import Filtered Data from MongoDB" on page 7-5

"Import Large Data from MongoDB" on page 7-8

"Database Toolbox Interface for MongoDB Error Messages" on page 7-21

External Websites

MongoDB Manual

Introduced in R2017b

dropCollection

Drop MongoDB collection

Syntax

```
dropCollection(conn, collection)
```

Description

`dropCollection(conn, collection)` drops an existing collection from MongoDB by using the MongoDB connection.

Examples

Drop Collection from MongoDB

Connect to MongoDB and drop a collection.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server, port, dbname)
```

```
conn =
```

```
mongo with properties:
```

```
Database: 'mongotest'  
UserName: ''  
Server: {'dbtb01'}  
Port: 27017  
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
TotalDocuments: 23485919
```

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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 =
```

```
1x7 cell array
```

```
Columns 1 through 5
```

```
    {'airlinesmall'}    {'employee'}    {'largedata'}    {'nyctaxi'}    {'product'}
```

```
Columns 6 through 7
```

```
    {'restaurants'}    {'taxidata'}
```

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 =  
1x7 cell array  
Columns 1 through 5  
    {'airlinesmall'}    {'employee'}    {'largedata'}    {'nyctaxi'}    {'product'}  
Columns 6  
    {'restaurants'}
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB connection

mongo object

MongoDB connection, specified as a mongo object.

collection — Collection name

string scalar

Collection name, specified as a string scalar.

Example: "taxidata"

Data Types: string

See Also

`close` | `createCollection` | `find` | `isopen` | `mongo`

Topics

"Import and Analyze Data from MongoDB" on page 7-2

"Import Filtered Data from MongoDB" on page 7-5

“Import Large Data from MongoDB” on page 7-8

“Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

MongoDB Manual

Introduced in R2017b

count

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 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 and count the total number of documents in a collection.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server, port, dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
    Database: 'mongotest'
    UserName: ''
```

```
Server: {'dbtb01'}
Port: 27017
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection. There are 25 documents in the collection.

```
collection = "employee";
n = count(conn, collection)
```

```
n =
```

```
25
```

Close the MongoDB connection.

```
close(conn)
```

Count Documents in MongoDB Query

Connect to MongoDB 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`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
      Database: 'mongotest'
      UserName: ''
      Server: {'dbtb01'}
      Port: 27017
      CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
      TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 character vector that contains a JSON-style string. This query sets the `department` field equal to the value `Sales`.

```
mongoquery = '{"department":"Sales"}';
```

Use the MongoDB query on the `employee` collection to count the total number of employees who work in the Sales department. A total of four employees work in the Sales department.

```
collection = "employee";  
n = count(conn, collection, 'Query', mongoquery)
```

```
n =
```

```
    4
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB connection

mongo object

MongoDB connection, specified as a mongo 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: `char` | `string`

Output Arguments

n — Total number of documents

numeric scalar

Total number of documents in a MongoDB collection or query, returned as a numeric scalar.

See Also

`close` | `distinct` | `find` | `isopen` | `mongo`

Topics

“Import and Analyze Data from MongoDB” on page 7-2

“Import Filtered Data from MongoDB” on page 7-5

“Import Large Data from MongoDB” on page 7-8

“Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

MongoDB Manual

Introduced in R2017b

distinct

Retrieve distinct values for field in MongoDB collection

Syntax

```
values = distinct(conn,collection,field)
values = distinct(conn,collection,field,'Query',mongoquery)
```

Description

`values = distinct(conn,collection,field)` returns distinct values for a field in a collection by using the MongoDB connection.

`values = distinct(conn,collection,field,'Query',mongoquery)` returns distinct values for a field in an executed MongoDB query on a collection.

Examples

Retrieve Distinct Values for Field in Collection

Connect to MongoDB and retrieve a distinct set of values for a field in a collection of documents. Here, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
    Database: 'mongotest'
```

```
UserName: ''
Server: {'dbtb01'}
Port: 27017
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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.

Retrieve a distinct set of values for a field in a document collection. Here, retrieve distinct salaries for all employees. `values` is a cell array of doubles.

```
collection = "employee";
field = "salary";
values = distinct(conn, collection, field);
```

Display the first three salaries in the cell array.

```
values{1:3}
```

```
ans =
```

```
60000
```

```
ans =  
    50000
```

```
ans =  
    55000
```

Close the MongoDB connection.

```
close(conn)
```

Retrieve Distinct Values for Field in MongoDB Query

Connect to MongoDB and retrieve a distinct set of values for a field in a MongoDB query. Here, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
    Database: 'mongotest'  
    UserName: ''  
    Server: {'dbtb01'}  
    Port: 27017  
    CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
    TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 character vector that contains a JSON-style string. This query sets the `department` field equal to the value `Sales`.

```
mongoquery = '{"department":"Sales"}';
```

Use the MongoDB query on the `employee` collection to retrieve a distinct set of values for a field. Here, retrieve distinct salaries of all employees in the Sales department. `values` is a cell array of doubles.

```
collection = "employee";
```

```
field = "salary";
```

```
values = distinct(conn,collection,field,'Query',mongoquery)
```

```
values =
```

```
    1×3 cell array
```

```
    {[60000]}    {[64440]}    {[66000]}
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB connection

mongo object

MongoDB connection, specified as a mongo object.

collection — Collection name

string scalar

Collection name, specified as a string scalar.

Example: "taxidata"

Data Types: string

field — Field

string scalar

Field in a collection, specified as a string scalar.

Example: "department"

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: char | string

Output Arguments

values — Distinct values

cell array

Distinct values of a field in a MongoDB collection or query, specified as a cell array. The cell array can contain numeric scalars for numeric data, character vectors for text data, and structures for nested documents.

See Also

close | count | find | isopen | mongo

Topics

“Import and Analyze Data from MongoDB” on page 7-2

“Import Filtered Data from MongoDB” on page 7-5

“Import Large Data from MongoDB” on page 7-8

“Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

MongoDB Manual

Introduced in R2017b

find

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 connection.

`documents = find(conn,collection,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'Limit',10` limits the number of documents returned to 10.

Examples

Retrieve All Documents in Collection

Connect to MongoDB, retrieve all documents in a collection, and import them into MATLAB. Here, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)
```

```
conn =
```

```
mongo with properties:
```

```
Database: 'mongotest'  
UserName: ''  
Server: {'dbtb01'}  
Port: 27017  
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection for document retrieval. Retrieve all documents in the collection by using the MongoDB connection. `documents` is a structure array.

```
collection = "employee";  
documents = find(conn, collection);
```

Display the first document in the collection. Each document is a structure with these fields.

Field	Description	Data Type
<code>x_id</code>	Unique identifier	Structure

Field	Description	Data Type
department	Department name	Character vector
employee	Employee identifier	Double
salary	Employee salary	Double

```
documents(1)
```

```
ans =
```

```
    struct with fields:
```

```
        x_id: [1x1 struct]
    department: 'Sales'
      employee: 1
        salary: 60000
```

Close the MongoDB connection.

```
close(conn)
```

Retrieve All Documents in MongoDB Query

Connect to MongoDB, retrieve all documents in a MongoDB query on a collection in the database, and import them into MATLAB. Here, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)
```

```
conn =
```

```
    mongo with properties:
```

```
        Database: 'mongotest'
        UserName: ''
        Server: {'dbtb01'}
        Port: 27017
    CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
    TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection for document retrieval. Create the MongoDB query as a character vector that contains a JSON-style string. This query retrieves all employees in the Sales department.

```
collection = "employee";  
mongoquery = '{"department":"Sales"}';
```

Retrieve all documents in the MongoDB query on the `employee` collection by using the MongoDB connection. `documents` is a structure 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 and retrieve documents in a MongoDB query on a collection in the database. Then, sort the results by a field in the documents. Here, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
      Database: 'mongotest'
      UserName: ''
      Server: {'dbtb01'}
      Port: 27017
      CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
      TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection for document retrieval. Create the MongoDB query as a character vector that contains a JSON-style string. This query retrieves all employees in the Sales department.

```
collection = "employee";  
mongoquery = '{"department":"Sales"}';
```

Create the sort query as a character vector 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 `employee` collection by using the MongoDB connection, and sort the documents. `documents` is a structure 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 sorted salaries for the first two employees.

```
documents(1:2).salary
```

```
ans =
```

```
60000
```

```
ans =
```

```
64440
```

Close the MongoDB connection.

```
close(conn)
```

Retrieve Specific Fields in Collection

Connect to MongoDB and retrieve all documents in a collection. Specify the fields to retrieve for each document. Import the documents into MATLAB. Here, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)
```

```
conn =
```

```
mongo with properties:
```

```
Database: 'mongotest'
UserName: ''
Server: {'dbtb01'}
Port: 27017
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection for document retrieval. Specify the fields to retrieve for each document by using a character vector that contains a JSON-style string. Here, return the `department` and `salary` fields.

```
collection = "employee";  
fields = '{"department":1.0,"salary":1.0}';
```

Retrieve all documents in the collection. Use the name-value pair argument `'Projection'` to retrieve the specified fields for each document. `documents` is a structure 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, `department` and `salary`.

```
documents(1)
```

```
ans =
```

```
    struct with fields:  
        x_id: [1×1 struct]  
    department: 'Sales'  
        salary: 60000
```

Close the MongoDB connection.

```
close(conn)
```

Retrieve Specific Number of Documents Using Offset

Connect to MongoDB 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. Here, each document in the collection represents an employee.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.


```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)

conn =
  mongo with properties:
    Database: 'mongotest'
    UserName: ''
    Server: {'dbtb01'}
    Port: 27017
    CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
    TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection for document retrieval.

```
collection = "employee";
```

Use the name-value pair argument 'Skip' to skip the first five documents in the collection. Then, use the name-value pair argument 'Limit' to return the next 10 documents in the collection. `documents` is structure array that contains 10 documents.

```
documents = find(conn,collection,'Skip',5,'Limit',10);
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB connection

mongo object

MongoDB connection, specified as a mongo object.

collection — Collection name

string scalar

Collection name, specified as a string scalar.

Example: "taxidata"

Data Types: string

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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 the comma-separated pair consisting of 'Query' and 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 the comma-separated pair consisting of 'Projection' and 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 the comma-separated pair consisting of 'Sort' and 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 the comma-separated pair consisting of 'Skip' and 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 the comma-separated pair consisting of 'Limit' and 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

Tips

- The `find` function estimates memory requirements when retrieving many documents using the Java heap. To avoid out-of-memory issues, the function automatically limits the number of returned documents in a single execution.

When an issue occurs, the function throws a warning message, for example, `Warning: Available memory is less than Total memory required. Limiting the RESULTSET from 15837001 to 59248. Use Skip and Limit to retrieve resultset in batches.`

To retrieve many documents, retrieve them in batches. For an example, see “Import Large Data from MongoDB” on page 7-8.

See Also

`close` | `insert` | `isopen` | `mongo` | `remove` | `update`

Topics

“Import and Analyze Data from MongoDB” on page 7-2

“Import Filtered Data from MongoDB” on page 7-5

“Import Large Data from MongoDB” on page 7-8

“Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

MongoDB Manual

Introduced in R2017b

insert

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 connection. Specify one or multiple documents to insert.

Examples

Insert One Document into Collection as Structure

Connect to MongoDB and export one document from MATLAB and insert it into a collection. Specify the document to insert as a structure. Here, the collection represents employee data.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
mongo with properties:
```

```
Database: 'mongotest'  
UserName: ''  
Server: {'dbtb01'}  
Port: 27017
```

```
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 these fields: `employee`, `department`, and `salary`.

```
document.employee = 28;
document.department = 'Sales';
document.salary = 200000;
```

Specify the `employee` collection. Insert the document into the collection by using the MongoDB connection. The `insert` function inserts one document into the collection.

```
collection = "employee";
n = insert(conn, collection, document)
```

```
n =
```

```
1
```

Close the MongoDB connection.

```
close(conn)
```

Insert Multiple Documents into Collection as Structure Array

Connect to MongoDB and export multiple documents from MATLAB and insert them into a collection. Specify documents to insert as a structure array. Here, the collection represents employee data.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
mongo with properties:
```

```
    Database: 'mongotest'  
    UserName: ''  
    Server: {'dbtb01'}  
    Port: 27017  
    CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
    TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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`, `department`, and `salary`. Create the documents structure array from these documents.

```
employee1.employee = 26;  
employee1.department = 'Sales';  
employee1.salary = 100000;  
  
employee2.employee = 27;  
employee2.department = 'Training';  
employee2.salary = 150000;  
  
documents = [employee1 employee2];
```

Specify the `employee` collection. Insert documents into the collection using the MongoDB connection. The `insert` function inserts two documents into the collection.

```
collection = "employee";  
n = insert(conn, collection, documents)  
  
n =  
  
     2
```

Close the MongoDB connection.

```
close(conn)
```

Insert Multiple Documents into Collection as Table

Connect to MongoDB and export documents from MATLAB and insert them into a collection. Specify documents to insert as a table. Here, the collection represents employee data.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)

conn =
  mongo with properties:
    Database: 'mongotest'
    UserName: ''
    Server: {'dbtb01'}
    Port: 27017
    CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
    TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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:

- `employee` — Double array
- `department` — Cell array of character vectors
- `salary` — Double array

Create the `documents` table from these workspace variables.

```
employee = [26;27];  
department = {'Sales';'Training'};  
salary = [100000;150000];  
documents = table(department,employee,salary);
```

Specify the `employee` collection. Insert documents into the collection using the MongoDB connection. The `insert` function inserts two documents into the collection.

```
collection = "employee";  
n = insert(conn,collection,documents)
```

```
n =
```

```
    2
```

Close the MongoDB connection.

```
close(conn)
```

Insert Multiple Documents into Collection as Cell Array of Structures

Connect to MongoDB and export documents from MATLAB and insert them into a collection. Specify documents to insert as a cell array of structures. Here, the collection represents employee data.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server,port,dbname)
```

```
conn =
```

```
    mongo with properties:
```

```
Database: 'mongotest'  
UserName: ''  
Server: {'dbtb01'}  
Port: 27017  
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 = 'Sales';  
employee1.employee = 26;  
employee1.salary = 100000;
```

```
employee2.department = 'Training';  
employee2.employee = 27;  
employee2.salary = 150000;
```

```
documents = {employee1;employee2};
```

Specify the `employee` collection. Insert documents into the collection using the MongoDB connection. The `insert` function inserts two documents into the collection.

```
collection = "employee";  
n = insert(conn, collection, documents)
```

```
n =
```

```
    2
```

Close the MongoDB connection.

```
close(conn)
```

Insert Map Object into Collection

Connect to MongoDB and export a `Map` object from MATLAB and insert it into a collection. Here, the collection represents employee data.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server, port, dbname)
```

```
conn =
```

```
    mongo with properties:
```

```
      Database: 'mongotest'  
      Username: ''  
      Server: {'dbtb01'}  
      Port: 27017  
      CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}  
      TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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.

Construct a Map object document that contains an employee's payroll data for the first three months of the year.

```
months = {'January', 'February', 'March'};
payslips = [4500, 5000, 4500];
document = containers.Map(months, payslips);
```

Specify the `employee` collection. Insert the Map object into the collection using the MongoDB connection. The `insert` function inserts one document into the collection.

```
collection = "employee";
n = insert(conn, collection, document)
```

```
n =
```

```
  1
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB connection

mongo object

MongoDB connection, specified as a mongo 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
- Map object
- 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" (MATLAB).

Output Arguments

n — Number of documents inserted

numeric scalar

Number of documents inserted into a collection in the database, returned as a numeric scalar.

See Also

`close` | `containers.Map` | `isopen` | `mongo` | `remove` | `update`

Topics

“Export MATLAB Data into MongoDB” on page 7-11

“Import and Export MATLAB Objects Using MongoDB” on page 7-15

“Import and Analyze Data from MongoDB” on page 7-2

“Import Filtered Data from MongoDB” on page 7-5

“Import Large Data from MongoDB” on page 7-8

“Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

MongoDB Manual

Introduced in R2017b

remove

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 connection. Use a MongoDB query to specify removing one or multiple documents.

Examples

Remove Documents Using MongoDB Query

Connect to MongoDB and remove documents from a collection. Specify a MongoDB query to determine which documents to remove. Here, the collection represents employee data.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)
```

```
conn =
```

```
mongo with properties:
```

```
Database: 'mongotest'
UserName: ''
Server: {'dbtb01'}
Port: 27017
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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. Here, specify the `employee` collection. Create the MongoDB query to identify documents in the Sales department.

```
collection = "employee";  
mongoquery = '{"department":"Sales"}';
```

Remove documents using the MongoDB query. The `remove` function removes six documents from the collection.

```
n = remove(conn, collection, mongoquery)
```

```
n =
```

```
  6
```

Close the MongoDB connection.

```
close(conn)
```

Remove All Documents from Collection

Connect to MongoDB and remove all documents from a collection.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number `27017`.

```
server = "dbtb01";
port = 27017;
dbname = "mongotest";
conn = mongo(server,port,dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
      Database: 'mongotest'
      UserName: ''
      Server: {'dbtb01'}
      Port: 27017
      CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
      TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection. Use an empty MongoDB query to specify removing all documents. The `remove` function removes three documents from the collection.

```
collection = "employee";  
n = remove(conn, collection, "{}")
```

n =

3

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB connection

mongo object

MongoDB connection, specified as a mongo 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: `char` | `string`

Output Arguments

n — Number of documents removed

numeric scalar

Number of documents removed from a collection in the database, returned as a numeric scalar.

See Also

`close` | `insert` | `isopen` | `mongo` | `update`

Topics

“Export MATLAB Data into MongoDB” on page 7-11

“Import and Export MATLAB Objects Using MongoDB” on page 7-15

“Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

MongoDB Manual

Introduced in R2017b

update

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 connection. Use MongoDB queries to find and update documents.

Examples

Update Documents in Collection

Connect to MongoDB 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. Here, the collection represents employee data.

Create a MongoDB connection to the database `mongotest`. Here, the database server `dbtb01` hosts this database using port number 27017.

```
server = "dbtb01";  
port = 27017;  
dbname = "mongotest";  
conn = mongo(server, port, dbname)
```

```
conn =
```

```
  mongo with properties:
```

```
    Database: 'mongotest'  
    UserName: ''  
    Server: {'dbtb01'}  
    Port: 27017
```

```
CollectionNames: {'airlinesmall', 'employee', 'largedata' ... and 3 more}
TotalDocuments: 23485919
```

`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 six document collections. The first three collection names are `airlinesmall`, `employee`, and `largedata`.
- This database contains 23,485,919 documents.

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 `employee` collection. Create a MongoDB query to find employees in the Sales department. Then, create a MongoDB query to increase the value in the `salary` field by 5000.

```
collection = "employee";
findquery = '{"department":"Sales"}';
updatequery = '{$inc:{"salary":5000}}';
```

Increase the salaries for all employees in the Sales department using the MongoDB connection. The `update` function updates seven documents in the collection.

```
n = update(conn, collection, findquery, updatequery)
```

```
n =
```

```
7
```

Close the MongoDB connection.

```
close(conn)
```

Input Arguments

conn — MongoDB connection

mongo object

MongoDB connection, specified as a mongo 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.

See Also

`close` | `insert` | `isopen` | `mongo` | `remove`

Topics

“Export MATLAB Data into MongoDB” on page 7-11

“Import and Export MATLAB Objects Using MongoDB” on page 7-15

“Database Toolbox Interface for MongoDB Error Messages” on page 7-21

External Websites

MongoDB Manual

Introduced in R2017b

