Database Toolbox™

User's Guide

MATLAB®

R2016b

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

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)

Contents

1 Before You Begin

2 Getting Started with Database Toolbox

3 Working with Data Sources

4 Using Database Explorer

5 Using Visual Query Builder

6 Using Database Toolbox Functions

7 Neo4j Topics

8 Functions — Alphabetical List

1

Before You Begin

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 standard ODBC-compliant and JDBC-compliant databases, including Oracle®, SAS®, MySQL®, Sybase®, 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.

Key Features

- Database Explorer app for working with relational databases interactively
- Support for graph database Neo4j and NoSQL database MongoDB
- 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 array

Working with Databases

Connecting to Databases

Before you can use this toolbox to connect to a database, you must set up the data sources. For details, see "Configuring a Driver and Data Source" on page 2-16.

Platform Support

This toolbox runs on all platforms that the MATLAB® software supports.

For details, see Database Toolbox system requirements at http://www.mathworks.com/products/database/requirements.html.

Note: This toolbox does not support running MATLAB software sessions with the `-nojvm` startup option enabled on UNIX® platforms. (UNIX is a registered trademark of The Open Group in the United States and other countries.)

Database Support

This toolbox supports importing and exporting data from any ODBC- and/or JDBC-compliant database management system, including:

- IBM DB2®
- IBM® Informix®
- Ingres®

- Microsoft Access
- Microsoft Excel®
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL (Postgres)
- Sybase SQL Anywhere®
- Sybase SQL Server®

If you are upgrading an earlier version of a database, you need not do anything special for this toolbox. Simply configure the data sources for the new version of the database application as you did for the original version.

Driver Support

This toolbox requires a database driver. Typically, you install a driver when you install a database. For instructions about how to install a database driver, consult your database administrator.

On Microsoft Windows® platforms, the toolbox supports Open Database Connectivity (ODBC) drivers and Oracle Java® Database Connectivity (JDBC) drivers.

Note: If you receive this message:

```
Invalid string or buffer length.
```

you might be using the wrong driver.

The JDBC/ODBC bridge is known to have issues with 64-bit database systems. Use a JDBC driver or the native ODBC interface to connect to these databases.

On UNIX platforms, the toolbox supports Java Database Connectivity (JDBC) drivers. If your database does not ship with JDBC drivers, download drivers from the Oracle JDBC Web site at `http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html`.

Structured Query Language (SQL)

This toolbox supports American National Standards Institute (ANSI®) standard SQL commands.

Data Type Support

You can import the following data types into the MATLAB Workspace and export them back to your database:

- `BOOLEAN`
- `CHAR`
- `DATE`
- `DECIMAL`
- `DOUBLE`
- `FLOAT`
- `INTEGER`
- `LONGCHAR`
- `NUMERIC`
- `REAL`
- `SMALLINT`
- `TIME`
- `TIMESTAMP`

  Note: When importing `TIMESTAMP` data into MATLAB, you might get an incorrect value near the daylight savings time change. Possible workarounds are to convert `TIMESTAMP` data to strings in your SQL query, and then convert them back to your desired type in MATLAB, or try using a different driver for your database.

- `TINYINT`

  Note: Database Toolbox interprets the `TINYINT` data type as `BOOLEAN` and imports it into the MATLAB workspace as logical `true (1)` or `false (0)`. For details about how Database Toolbox handles `BOOLEAN` data, see "Importing and Exporting Boolean Data" on page 5-58.

- `VARCHAR`
- `NTEXT`

You can import data of types not included in this list into the MATLAB Workspace. However, you might need to manipulate such data before you can process it in MATLAB.

Note: Data types `LONGCHAR` and `NTEXT` are not supported for the native ODBC interface.

For Microsoft SQL Server, the data type `uniqueidentifier` is not supported for the native ODBC interface.

Data Retrieval Restrictions

Spaces in Table Names or Column Names

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

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`.

More About

- "Creating and Running SQL Queries" on page 1-9

Creating and Running SQL Queries

You can select data from your database and import it into MATLAB by doing any of the following:

- Use Database Explorer or the command line.
- Write queries using SQL.
- Use MATLAB to generate the SQL.

Then, if you want to repeat your tasks, then automate them by generating a MATLAB script.

Writing a query requires knowledge of SQL and experience using the command line. Use the `exec` function to write SQL if you have short or simple SQL queries that are easy to write as a character vector. Also, use the `exec` function to add MATLAB variables to your SQL query. If you have a long SQL query or multiple SQL queries that you want to run sequentially, create an SQL script file containing your SQL queries. To execute the SQL script file, use `runsqlscript`.

If you are unfamiliar with writing SQL code, then you can use Database Explorer to create SQL queries. For details, see “Define Query Criteria to Refine Results” on page 4-21. After creating the query using Database Explorer, you can generate the SQL for this query. For details, see “Save Queries as SQL Code” on page 4-25. You can embed the generated SQL into the SQL query that you specify in `exec`. Or, you can create an SQL script file to use with the `runsqlscript` function.

If you want to automate the current task after the SQL is created, then generate a MATLAB script. For details, see “Generate MATLAB Code” on page 4-26.

More About

- “Managing Memory to Import Data” on page 6-50
- “Data Retrieval Restrictions” on page 1-8
- “Data Type Support” on page 1-6

2

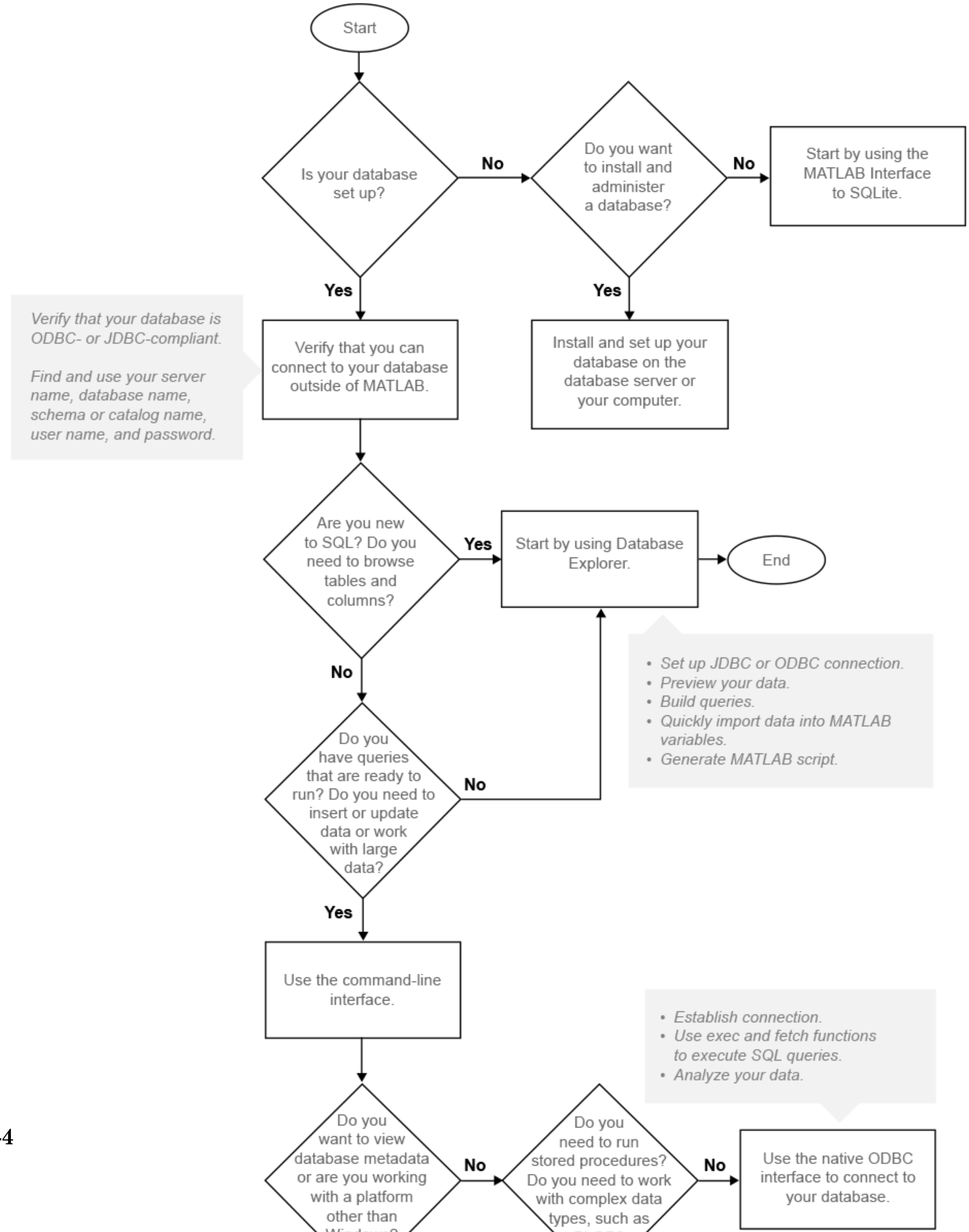
Getting Started with Database Toolbox

Working with a Database and MATLAB

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

1. 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 the 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 whether you want to use Database Explorer or the command line using the following flow chart.
4. Choose whether you want to use an ODBC or 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. Define your data source for ODBC-compliant drivers or add the full path of the driver to the static Java class path for JDBC-compliant drivers. For details, see "Configuring a Driver and Data Source" on page 2-16.
7. Test the connection to your database using Database Explorer or the command line.
8. Connect to your database using Database Explorer or the command line. For details, see "Connecting to a Database" on page 2-191.
9. Select data from your database and import the data into a MATLAB variable using Database Explorer or the command-line `exec` and `fetch` functions. For details, see "Selecting Data" on page 2-195.
10. Insert data into your database by exporting data from a MATLAB variable using `datainsert`, `fastinsert`, and `insert` functions. For details, see "Inserting Data Using the Command Line" on page 2-197.
11. To automate your tasks using the Database Explorer import functionality, generate a MATLAB script. For details, see "Generate MATLAB Code" on page 4-26.

For a graphical representation of the steps and the decisions you must make, see this flow chart.


Start
Is your database set up?
No
Do you want to install and administer a database?
No
Start by using the MATLAB Interface to SQLite.
Yes
Yes
Verify that your database is ODBC- or JDBC-compliant.
Find and use your server name, database name, schema or catalog name, user name, and password.
Verify that you can connect to your database outside of MATLAB.
Install and set up your database on the database server or your computer.
Are you new to SQL? Do you need to browse tables and columns?
Yes
Start by using Database Explorer.
End
• Set up JDBC or ODBC connection.
• Preview your data.
• Build queries.
• Quickly import data into MATLAB variables.
• Generate MATLAB script.
No
Do you have queries that are ready to run? Do you need to insert or update data or work with large data?
No
Yes
Use the command-line interface.
• Establish connection.
• Use exec and fetch functions to execute SQL queries.
• Analyze your data.
Do you want to view database metadata or are you working with a platform other than
No
Do you need to run stored procedures? Do you need to work with complex data types, such as
No
Use the native ODBC interface to connect to your database.

More About

- "Initial Setup Requirements" on page 2-12
- "Working with Database Explorer" on page 4-2
- "Choosing Between ODBC and JDBC Drivers" on page 2-13
- "Configuring a Driver and Data Source" on page 2-16
- "Connecting to a Database" on page 2-191
- "Working with the MATLAB Interface to SQLite" on page 2-6

Working with the MATLAB Interface to SQLite

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 a Driver and Data Source" on page 2-16.

In this section...

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
Database installation	Not required	Required

	SQLite Connection Using the MATLAB Interface to SQLite	SQLite Database Connection Using a JDBC Driver
Database administration	Not required	Required
Database connection function	`sqlite`	`database`
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 the MATLAB® Interface to SQLite” on page 6-75

More About

- “Working with a Database and MATLAB” on page 2-3
- “Configuring a Driver and Data Source” on page 2-16

External Websites

- SQLite Home Page

Connection Options

In this section...

There are various ways to connect to your database using Database Toolbox. If you have access to a database, create a data source. Then, you can connect to your database either by using Database Explorer 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 the MATLAB Interface to SQLite" on page 2-6.

Creating or Connecting to a Data Source

If you already have your 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 a Driver and Data Source" on page 2-16. Otherwise, see Driver Installation to help you install your driver. If your data sources are defined, then you are ready to connect to your database. If you created JDBC data sources using VQB, then see "Migrate from VQB to Database Explorer" on page 4-2. For details, see "Connecting to a Database" on page 2-191. Once connected, you can begin to explore your database using Database Explorer or the command line to view your data. For details, see "Selecting Data" on page 2-195.

Defining Operating System Authentication

Operating system authentication allows you to connect to your 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 easy 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 "Step 3. Set up the operating system authentication." on page 2-38

Connection Options

Use this table to choose your best connection option.

Connection Option	Why Use This Option?
Database Explorer	Use Database Explorer to: • Visually inspect the structure, or schema, of your database. • Assess the general size of your 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. For details, see "Selecting Data" on page 2-195.
Command line	Use the command line to: • 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.

Connection Option	Why Use This Option?
Native ODBC connection	Connect to your database with maximum performance. For details about the native ODBC interface, see "Connecting to a Database Using the Native ODBC Interface" on page 3-18.
JDBC connection	Achieve maximum platform independence. Use functionality not supported by native ODBC.

Connection Option	Why Use This Option?
ODBC connection	Only use this option after trying to connect to your database using the native ODBC and JDBC connections.
SQLite connection	Import data without installing a database or a driver. For details about the MATLAB interface to SQLite, see “Working with the MATLAB Interface to SQLite” on page 2-6.

Working with Multiple Databases

You can connect to multiple databases using Database Explorer or the command line. For details, see “Work with Multiple Databases” on page 4-17.

See Also

`database`

More About

- “Choosing Between ODBC and JDBC Drivers” on page 2-13
- “Connecting to a Database” on page 2-191
- “Connecting to a Database Using the Native ODBC Interface” on page 3-18
- “Selecting Data” on page 2-195
- “Working with the MATLAB Interface to SQLite” on page 2-6

Initial Setup Requirements

Refer to the following setup requirements to establish the first connection to your database.

- 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 the MATLAB Interface to SQLite" on page 2-6.
- If you use Visual Query Builder (VQB) to explore the data in your database, migrate to the Database Explorer app. For details, see "Migrate from VQB to Database Explorer" on page 4-2.
- 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 of where the JDBC driver is installed. For some JDBC drivers, you 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 needed for establishing connection to your database.
- Ensure that you have access to your database and driver documentation.
- Check 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 or SQLite database files. To set up connection to your database using operating system authentication from MATLAB, there can be additional required steps.
- Ensure that you have write access to the path MATLAB displays after executing `prefdir` on the command line.

See Also

`database`

More About

- "Working with a Database and MATLAB" on page 2-3
- "Connecting to a Database" on page 2-191

Choosing Between ODBC and JDBC Drivers

In this section...

Defining Database Drivers

Different database vendors, such as Microsoft or Oracle, implement their database systems using various technologies depending on customer needs, market demands, and several other factors. Software applications written in popular programming languages, such as C, C++, or 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 are simply standards, or a set of rules recommended for efficient communication with a database. The database vendor is responsible for implementing and providing drivers that are committed to 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++.

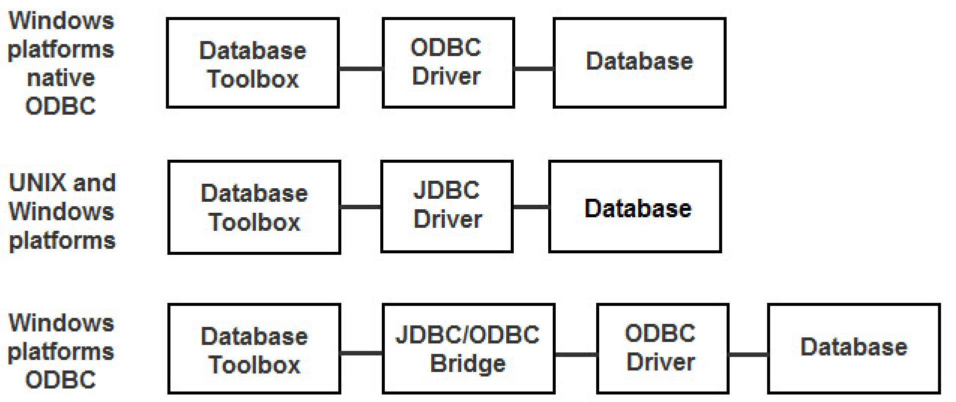
JDBC is a standard interface that enables communication between applications based on Oracle Java and database management systems.

The JDBC/ODBC bridge is a Java library that allows Java applications to access the ODBC interface.

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 or uses the JDBC/ODBC bridge to connect to an ODBC driver. The JDBC/ODBC bridge is automatically installed as part of the MATLAB JVM™.

Note: The JDBC/ODBC bridge functionality will be removed in a future release.

The following figure illustrates how drivers interact with Database Toolbox.



Depending on your environment and what you want to accomplish, decide whether using an ODBC driver or a JDBC driver suits your needs the best. Use the following to help you decide.

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
- Using Database Toolbox functions not supported by the native ODBC interface (such as `runstoredprocedure`)
- Working with complex or long data types (e.g., `LONG`, `BLOB`, text, etc.)

Tip: On Windows systems that support both ODBC and JDBC drivers, JDBC drivers and the native ODBC interface provide better connectivity and performance than the JDBC/ODBC bridge. First, use the native ODBC or JDBC drivers to connect to your database. Use the JDBC/ODBC bridge only after trying to connect through native ODBC or JDBC drivers.

For a list of native ODBC supported functionality and a full comparison of the JDBC/ODBC bridge to the native ODBC interface, see “Connecting to a Database Using the Native ODBC Interface” on page 3-18.

See Also

`close` | `database`

More About

- “Working with a Database and MATLAB” on page 2-3
- “Connection Options” on page 2-9
- “Configuring a Driver and Data Source” on page 2-16
- “Connecting to a Database” on page 2-191
- “Working with Large Data Sets” on page 2-199

Configuring a Driver and Data Source

To connect to an installed database, install the driver. Then, 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 the 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	Mac OS X 64-bit	Linux 64-bit
Microsoft Access	"Microsoft Access ODBC for Windows" on page 2-19		
Microsoft SQL Server	"Microsoft SQL Server ODBC for Windows" on page 2-26 "Microsoft SQL Server JDBC for Windows" on page 2-35	"Microsoft SQL Server JDBC for Mac OS X" on page 2-104	"Microsoft SQL Server JDBC for Linux" on page 2-111
Oracle	"Oracle ODBC for Windows" on page 2-46 "Oracle JDBC for Windows" on page 2-50	"Oracle JDBC for Mac OS X" on page 2-118	"Oracle JDBC for Linux" on page 2-125

Database	Platform		
	Windows	Mac OS X 64-bit	Linux 64-bit
MySQL	"MySQL ODBC for Windows" on page 2-59 "MySQL JDBC for Windows" on page 2-65	"MySQL JDBC for Mac OS X" on page 2-132	"MySQL JDBC for Linux" on page 2-139
PostgreSQL	"PostgreSQL ODBC for Windows" on page 2-71 "PostgreSQL JDBC for Windows" on page 2-77	"PostgreSQL JDBC for Mac OS X" on page 2-146	"PostgreSQL JDBC for Linux" on page 2-153
SQLite	"SQLite JDBC for Windows" on page 2-83	"SQLite JDBC for Mac OS X" on page 2-160	"SQLite JDBC for Linux" on page 2-167
Sybase	"Sybase ODBC for Windows" on page 2-90 "Sybase JDBC for Windows" on page 2-97	"Sybase JDBC for Mac OS X" on page 2-174	"Sybase JDBC for Linux" on page 2-181

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

For ODBC- or JDBC- compliant databases that are not listed in the table, see "Other ODBC- or JDBC-Compliant Databases" on page 2-188.

See Also

`close` | `database`

More About

- "Working with a Database and MATLAB" on page 2-3

- "Initial Setup Requirements" on page 2-12
- "Choosing Between ODBC and JDBC Drivers" on page 2-13
- "Connecting to a Database" on page 2-191
- "Working with the 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 your Microsoft Access database. This tutorial uses the Microsoft Access Driver (*.mdb, *accdb) to connect to the Microsoft Access 2010 database.

In this section...

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

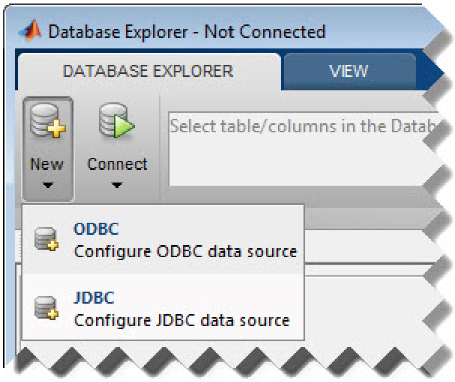
Step 2. Set up the data source using Database Explorer.

Set up your Microsoft Access database using Database Explorer. When setting up a data source for use with an ODBC driver, 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 U.S. English version of Microsoft Access 2010 for Windows systems.

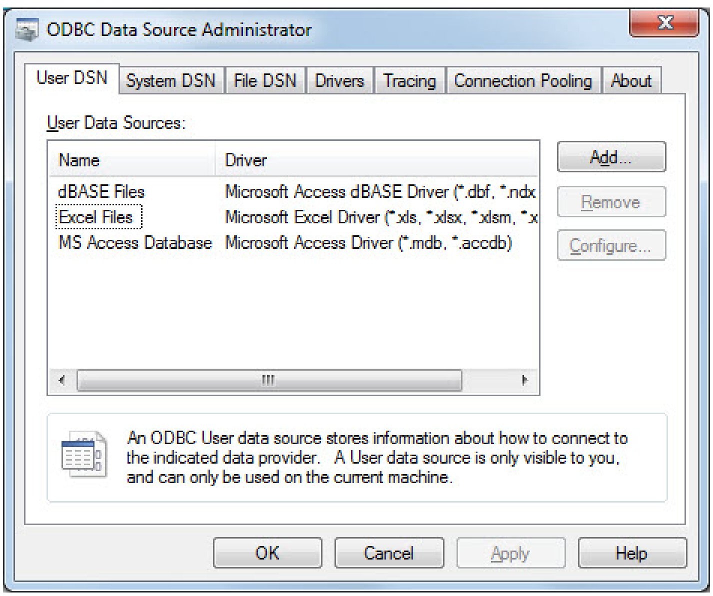
1. Close all open databases, including `tutorial.mdb`, in the database program.
2. Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting**

section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.

3 Click the **Database Explorer** tab, and then select **New** > **ODBC**.



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



4. Click the **User DSN** tab and then click **Add**. When setting up an ODBC data source, you can use a User DSN or System DSN. A User DSN is specific to the user on a machine. Any data sources a user defines under User DSN are seen only by that specific user. Conversely, a System DSN is not specific to the user on a machine. Any data sources a user defines under System DSN on a machine can be seen by any user who logs into that machine. Your ability to set up a User DSN or System DSN might depend on the database and ODBC driver you are using. For details, contact your database administrator or your database ODBC driver documentation.
5. A list of installed ODBC drivers appears in the Create New Data Source dialog box. Select `Microsoft Access Driver (*.mdb, *.accdb)` and click **Finish**.
6. In the ODBC Microsoft Access Setup dialog box for your driver, enter `dbtoolboxdemo` as the data source name. Enter `tutorial database` as the description. Click **Select** to open the Select Database dialog box.

7. Specify the database you want to use. For the `dbtoolboxdemo` data source, select `tutorial.mdb`. If your 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 that 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 `dbtoolboxdemo` and any additional data sources that you added in the **User DSN** tab. Click **OK** to close the dialog box.

9 Test the connection to the data source by using Database Explorer to connect to the database.

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

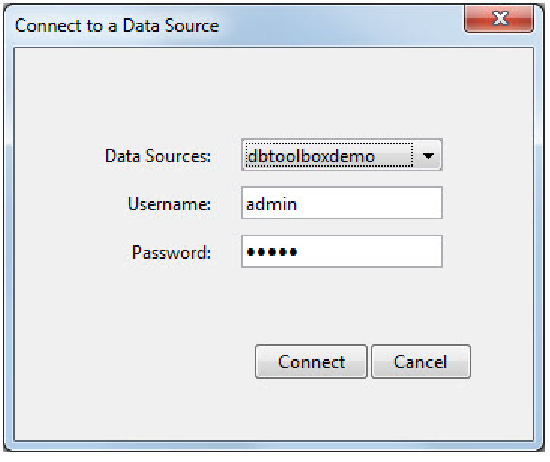
Step 3. Connect using Database Explorer or the command line.

Connect to Microsoft Access using Database Explorer.

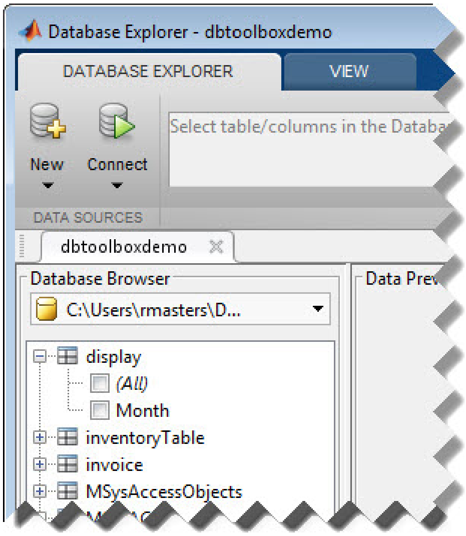
1 After setting up the data source, click **Connect** in the **Database Explorer** tab.

2 In the Connect to a Data Source dialog box, connect to your database by selecting the data source name `dbtoolboxdemo` from the **Data Sources** list.

3 Enter a user name and password and click **Connect**.

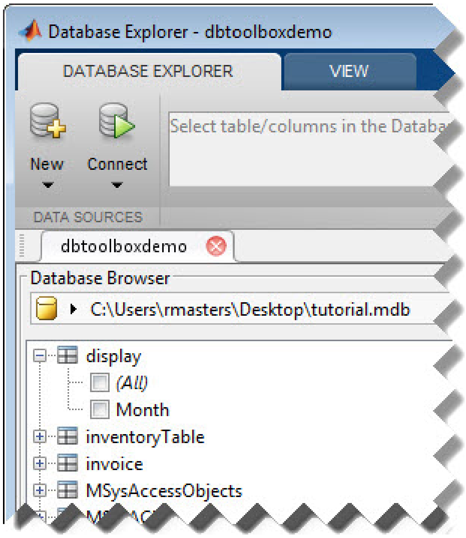


   Database Explorer connects to the database and displays the tables list, or database schema, on the left side of the window.



4 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **dbtoolboxdemo** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Microsoft Access using the native ODBC connection command line.

1 Connect to the database with the ODBC data source name. For example, the following code assumes you are connecting to a data source named `dbtoolboxdemo` with user name `admin` and password `admin`.

```
conn = database.ODBCConnection('dbtoolboxdemo','admin','admin');
```

2 Close the database connection `conn`.

```
close(conn)
```

Connect to Microsoft Access using the ODBC connection command line.

1 Connect to the database with the ODBC data source name. For example, the following code assumes you are connecting to a data source named `dbtoolboxdemo` with user name `admin` and password `admin`.

```
conn = database('dbtoolboxdemo','admin','admin');
```

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database`

More About

- “Working with Database Explorer” on page 4-2

Microsoft SQL Server ODBC for Windows

This tutorial shows how to set up a data source and connect to your Microsoft SQL Server database. This tutorial uses the Microsoft SQL Server Native Client 11.0 Driver to connect to the Microsoft SQL Server 2012 Express database.

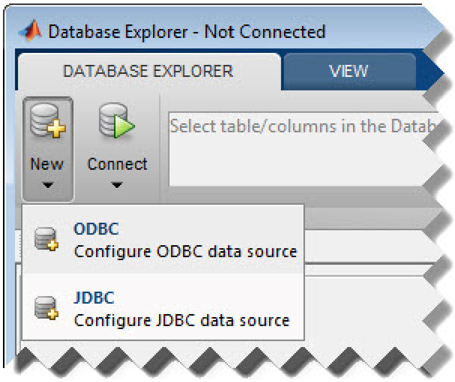
Step 1. Verify the driver installation.

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

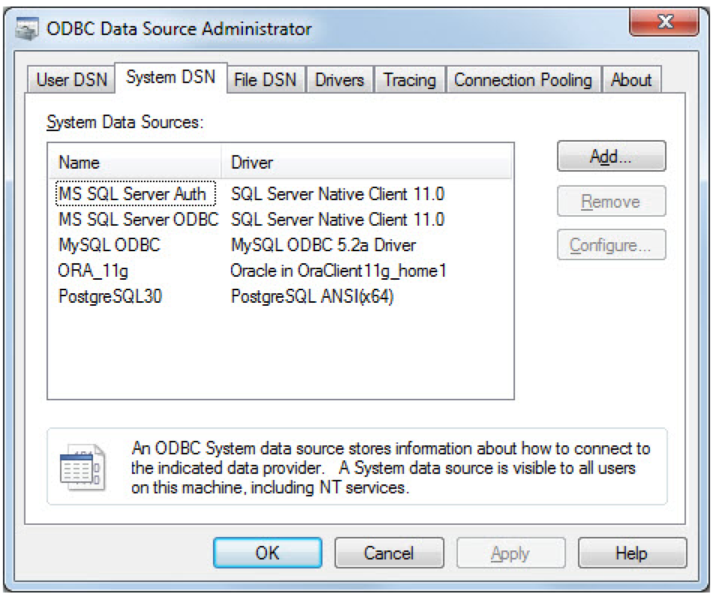
Note: The Database Toolbox no longer supports connection to a database using a 32-bit driver. Use a 64-bit version of Microsoft SQL Server. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see "Microsoft SQL Server JDBC for Windows" on page 2-35. For details about working with a 64-bit version of Windows, see http://www.mathworks.com/products/matlab/preparing-for-64-bit-windows.html.

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

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New > ODBC**.



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



3 Click the **System DSN** tab and then click **Add**. When setting up an ODBC data source, you can use a User DSN or System DSN. A User DSN is specific to the user on a machine. Any data sources a user defines under User DSN are seen only by that specific user. Conversely, a System DSN is not specific to the user on a machine. Any

data sources a user defines under System DSN on a machine can be seen by any user who logs into that machine. Your ability to set up a User DSN or System DSN might depend on the database and ODBC driver you are using. For details, contact your database administrator or your database ODBC driver documentation.

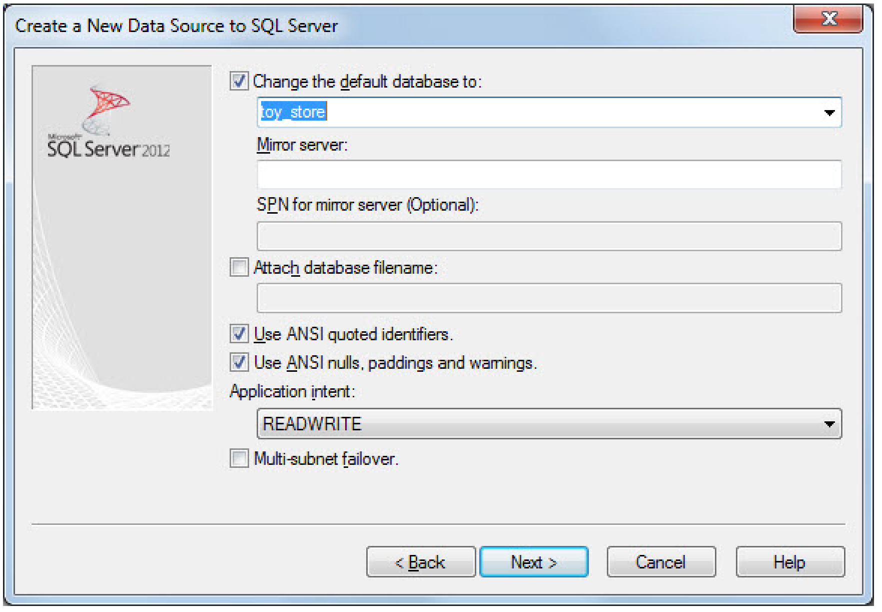
4 A list of installed ODBC drivers appears in the Create New Data Source dialog box. Select `SQL Server Native Client 11.0` and click **Finish**.

5 In the Create a New Data Source to SQL Server dialog box, enter an appropriate name for your data source. You use this name to establish a connection to your database. Here, enter `MS SQL Server` as the data source name in the **Name** field. Enter `Microsoft SQL Server` as the description in the **Description** field. Select the database server for this data source to use in the **Server** field. Consult your database administrator for the name of your database server. Click **Next**.

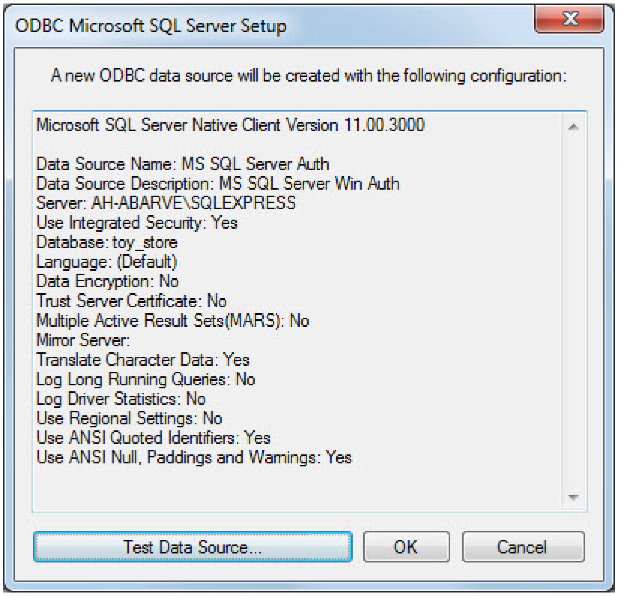
6 If you want to connect to Microsoft SQL Server using Windows authentication, click the **With Integrated Windows Authentication** option button. Then click **Next**.

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

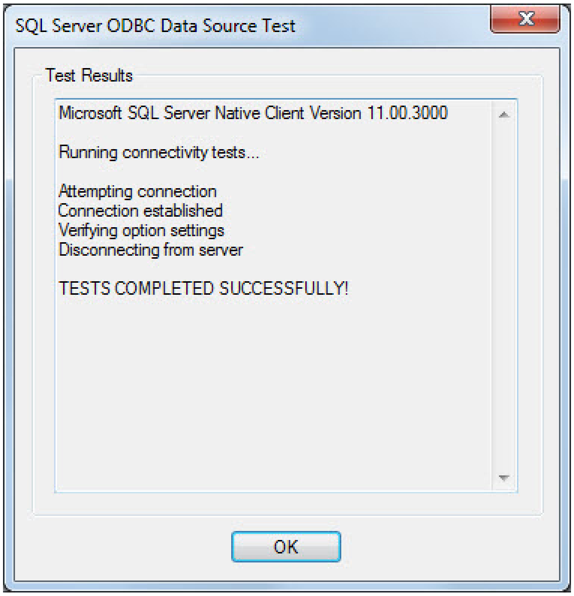
7 In the Create a New Data Source to SQL Server dialog box, select the **Change the default database to** check box and enter the name of the default database on the database server for connection. Here, use the database `toy_store`. Then click **Next**.



8. Here, 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 establishes successfully, this message appears in the SQL Server ODBC Data Source Test dialog box: 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 in 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 Microsoft SQL Server database using Database Explorer or the command line with the native ODBC connection.

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

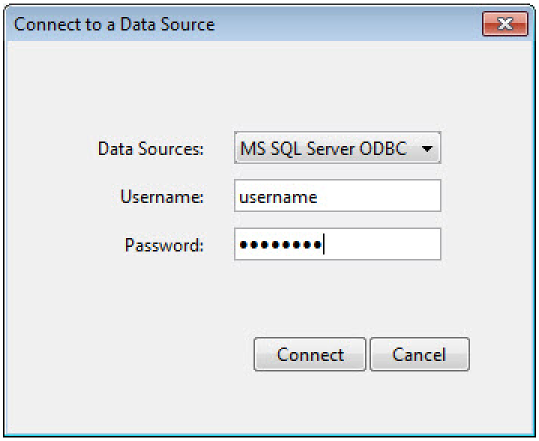
Connect to Microsoft SQL Server using Database Explorer.

If you experience issues connecting using Database Explorer, use the native ODBC interface with the command line or JDBC to connect to your database.

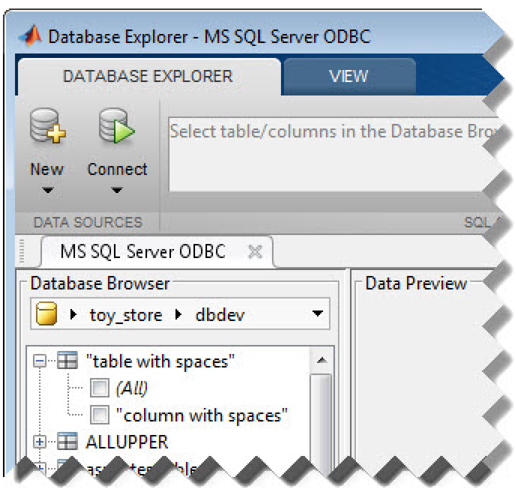
1 After setting up the data source, click **Connect** in the **Database Explorer** tab.

2 In the Connect to a Data Source dialog box, connect with operating system authentication by selecting the data source that you set up with Windows authentication from the **Data Sources** list. Leave the user name and password blank. Click **Connect**.

3 Connect without operating system authentication by selecting the data source that you set up without Windows authentication. Enter a user name and password. Click **Connect**.

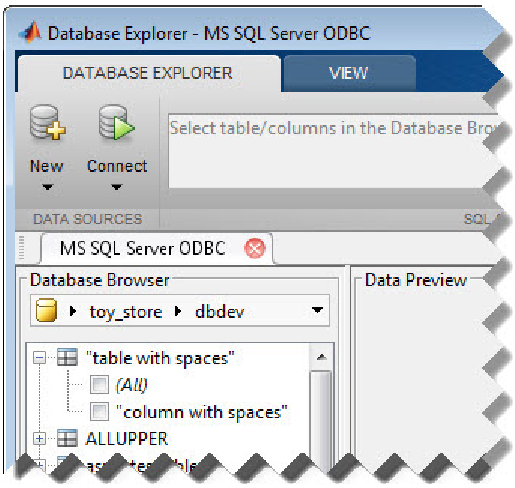


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



4 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **MS SQL Server ODBC** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Microsoft SQL Server using the native ODBC connection command line.

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

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

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

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

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database`

More About

- "Working with Database Explorer" on page 4-2

Microsoft SQL Server JDBC for Windows

This tutorial shows how to set up a data source and connect to your Microsoft SQL Server database. This tutorial uses the Microsoft JDBC Driver 4.0 for Microsoft SQL Server to connect to the Microsoft SQL Server 2012 Express database.

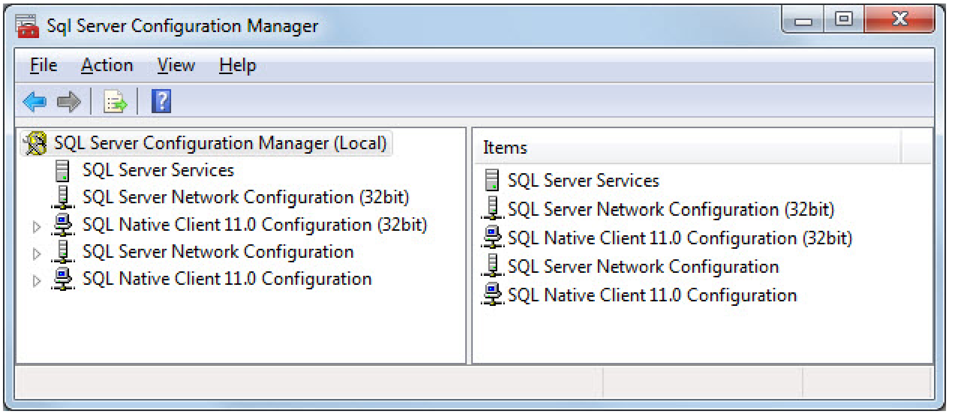
Step 1. Verify the driver installation.

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

Step 2. Verify the port number.

To connect to your database using a JDBC driver, you must know the port number. Use the following steps on the machine where Microsoft SQL Server is installed to find your port number. If you experience connection issues with the port number that you find, contact your database administrator.

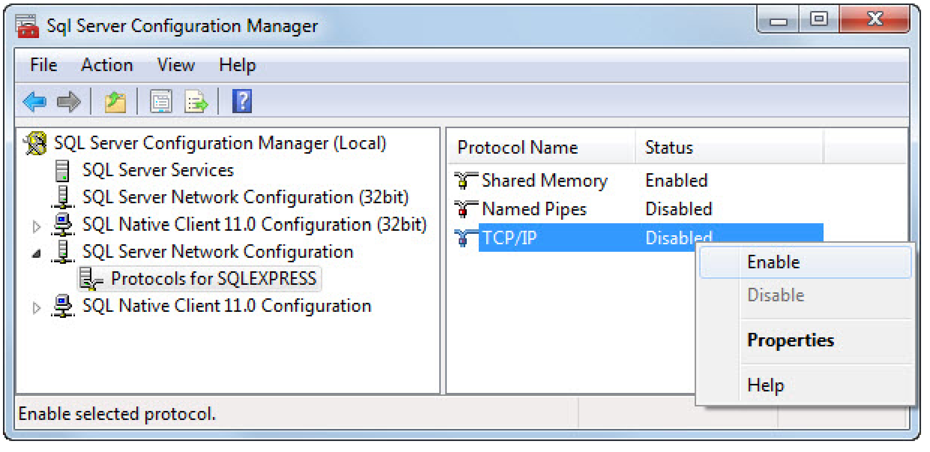
1 On the machine where your Microsoft SQL Server database is installed, click **Start**. Select your Microsoft SQL Server version folder and click **Configuration Tools**. Then click **SQL Server Configuration Manager**.



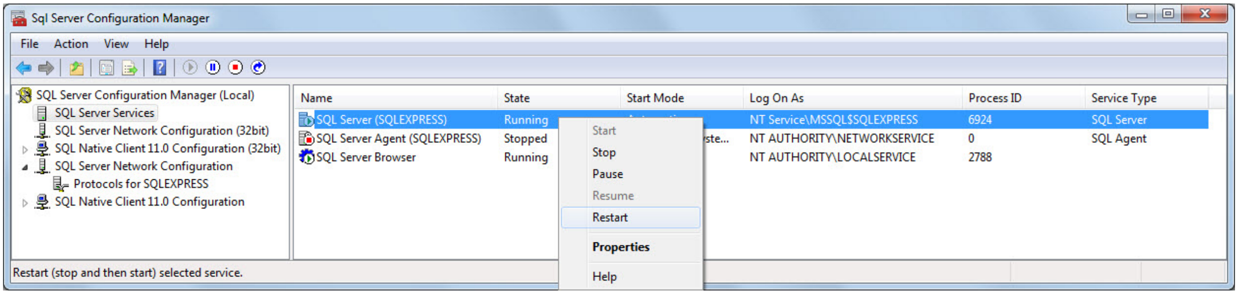
2 In the Sql Server Configuration Manager window, click **SQL Server Network Configuration** on the left side. Double-click **Protocols for SQLEXPRESS**.

3 See if TCP/IP is enabled. If so, skip the steps for enabling TCP/IP and restarting the server.

4 If TCP/IP is disabled, right-click **TCP/IP** and select **Enable**.

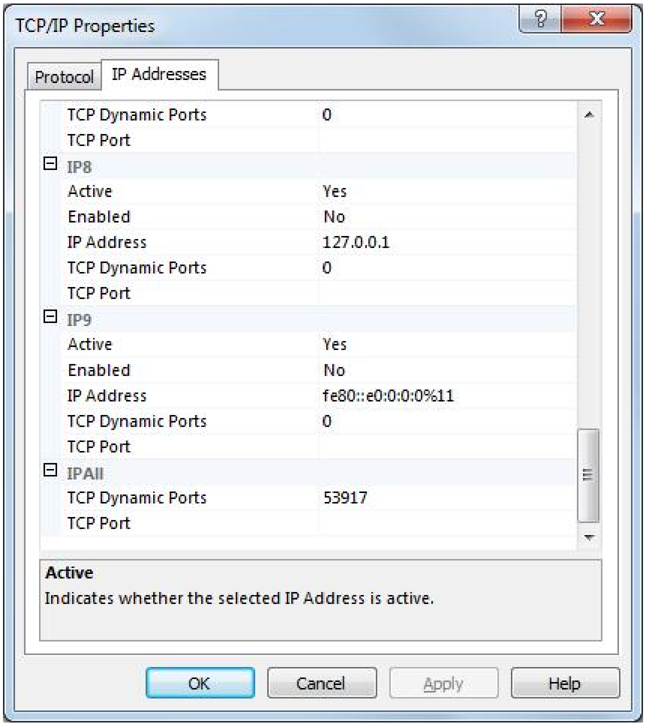


5 To finish the process of enabling the TCP/IP protocol, restart the server. Click **SQL Server Services** on the left side of the window. Right-click **SQL Server (SQLEXPRESS)** and click **Restart**.



6 The server restarts enabling TCP/IP. Click **Protocols for SQLEXPRESS** and right-click **TCP/IP**. Select **Properties**.

7 In the TCP/IP Properties dialog box, scroll to the bottom in the **IP Addresses** tab until you see **IP All** group. The number next to the **TCP Dynamic Ports** field is the port number. Use this port number in the JDBC connection parameters for Database Explorer or the command line. Here, the port number is `53917`. If this number is `0` or you want to configure your Microsoft SQL Server database server to listen to a specific port, delete the entry in the **TCP Dynamic Ports** field and enter another port number in the **TCP Port** field.



Step 3. Set up the operating system authentication.

Windows authentication lets you to connect to your 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 easy maintenance of database access credentials. After you add the required libraries to the system path, the Microsoft SQL Server JDBC driver allows 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 “Bring Java Classes into MATLAB Workspace”.

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` command in the Command Window. The output of this command is a file path to a folder on your computer.

3 Close MATLAB if it is running.
4 Navigate to the folder and create a file called `javalibrarypath.txt` in the folder.
5 Open `javalibrarypath.txt` and insert the path to the Java library file `sqljdbc_auth.dll`. Use the x64 folder. In the entry, include the full path to the library file. Do not include the library file name. For example, `C:\DB_Drivers\sqljdbc_4.0\enu\auth\x64`.

   The `sqljdbc_auth.dll` 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.
6 Open MATLAB.

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

1 Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2 Close MATLAB if it is running.
3 Navigate to the folder and create a file called `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 and the JAR file name. For example, `C:\DB_Drivers\sqljdbc_4.0\enu\sqljdbc4.jar`. Save and close `javaclasspath.txt`.
5 Restart MATLAB.

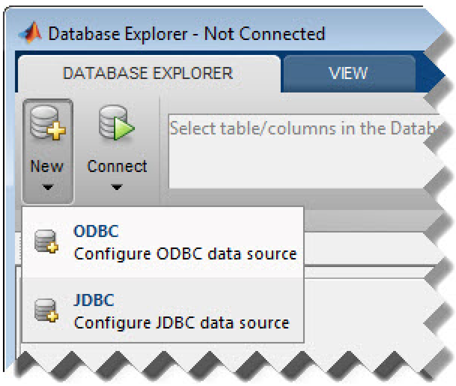
Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".

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

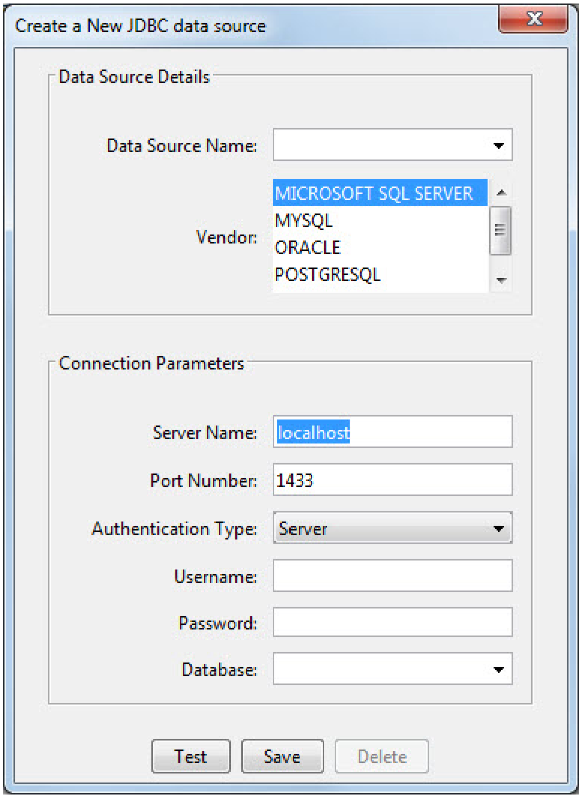
This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see "Connect to Microsoft SQL Server using the JDBC connection command line." on page 2-44

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.

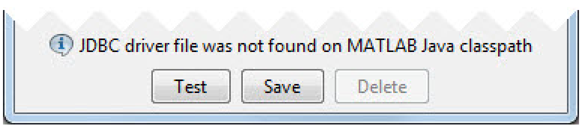
2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `MICROSOFT SQL SERVER` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 4.



4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field.

5 Create a data source without Windows authentication by setting the **Authentication Type** to `Server`.

   Or, create a data source with Windows authentication by setting the **Authentication Type** to `Windows` and leaving **Username** and **Password** blank.

6 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

7 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

8 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

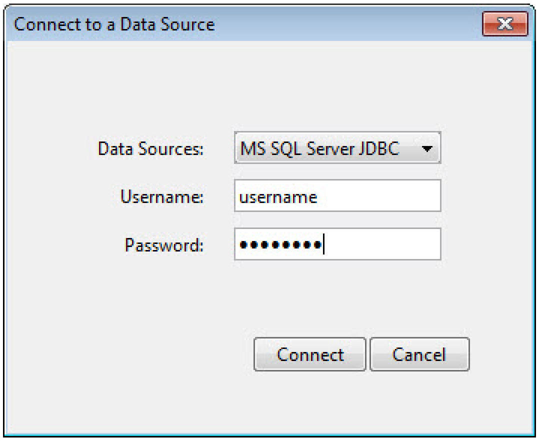
   After you complete the data source setup, connect to the Microsoft SQL Server database using Database Explorer or the command line with the JDBC connection.

Step 6. Connect using Database Explorer or the command line.

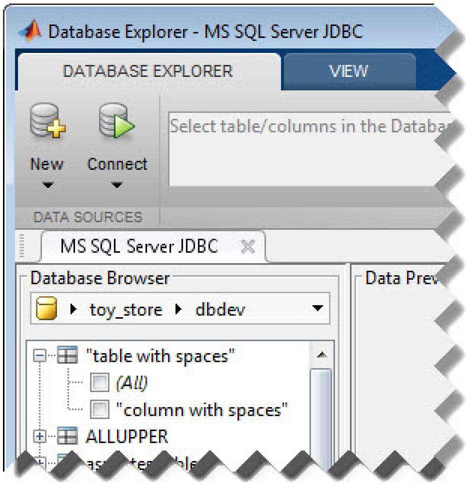
Connect to Microsoft SQL Server using Database Explorer.

1 After setting up the data source, connect with operating system authentication by selecting the data source that you set up with Windows authentication from the **Data Sources** list. Leave the user name and password blank. Click **Connect**.

2 Connect to your database without operating system authentication by selecting the data source that you set up without Windows authentication. Enter a user name and password. Click **Connect**.

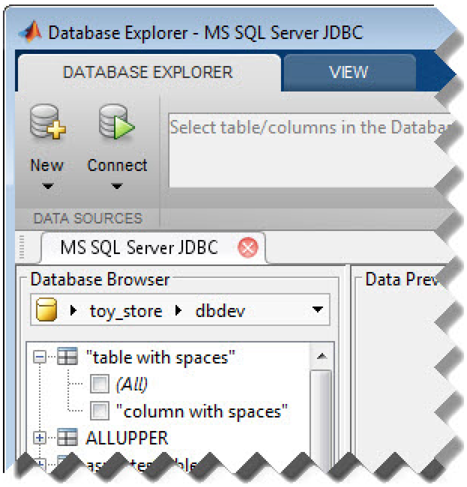


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



3 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **MS SQL Server JDBC** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Microsoft SQL Server using the JDBC connection command line.

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

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

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

Or, to connect without operating system authentication, use the `AuthType` name-value pair argument of `database` to specify a connection to the database server

`Server`. For example, the following code assumes you are connecting to a database named `dbname` with user name `username` and password `pwd`.

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

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2
- "Bring Java Classes into MATLAB Workspace"

Oracle ODBC for Windows

This tutorial shows how to set up a data source and connect to your Oracle database. This tutorial uses the OraClient11g_home1 ODBC driver to connect to the Oracle 11g Enterprise Edition database.

Step 1. Verify the driver installation.

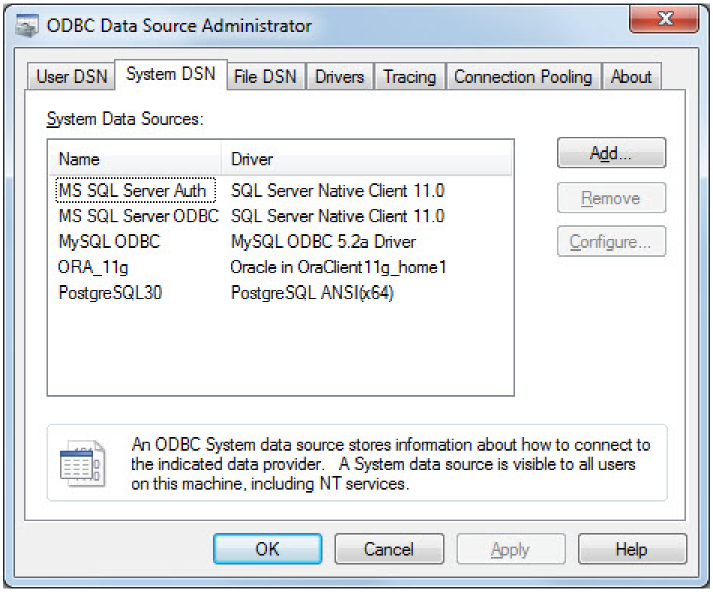
The ODBC driver is typically preinstalled on your computer. For details about the driver installation or troubleshooting the installation, contact your database administrator or refer to your database documentation on ODBC drivers. For information about the Microsoft ODBC Data Source Administrator, see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

Note: The Database Toolbox no longer supports connection to a database using a 32-bit driver. Use a 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 2-50. For details about working with a 64-bit version of Windows, see http://www.mathworks.com/products/matlab/preparing-for-64-bit-windows.html.

Step 2. Set up the data source using the ODBC Data Source Administrator.

Set up an ODBC data source using the ODBC Data Source Administrator for Oracle with or without Windows authentication. Database Explorer cannot work with the Oracle ODBC driver because of an issue with the JDBC/ODBC bridge. For details, see "Database Explorer Error Messages" on page 3-15.

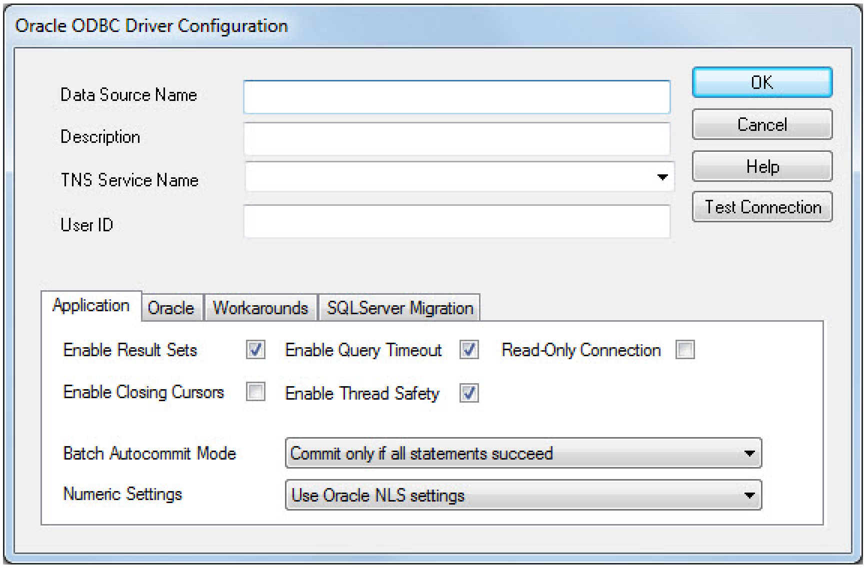
1 Click **Start**. Select **Administrative Tools > Data Sources (ODBC)** to define the ODBC data source. The ODBC Data Source Administrator dialog box opens. For details about locating this program on your computer, see Driver Installation.



2 Click the **System DSN** tab and then click **Add**. When setting up an ODBC data source, you can use a User DSN or System DSN. A User DSN is specific to the user on a machine. Any data sources a user defines under User DSN are seen only by that specific user. Conversely, a System DSN is not specific to the user on a machine. Any data sources a user defines under System DSN on a machine can be seen by any user who logs into that machine. Your ability to set up a User DSN or System DSN might depend on the database and ODBC driver you are using. For details, contact your database administrator or your database ODBC driver documentation.

3 A list of installed ODBC drivers appears in the Create New Data Source dialog box. Select the ODBC driver `Oracle in OraClient11g_home1`. Your ODBC driver might have a different name. Click **Finish**.

4 In the Oracle ODBC Driver Configuration dialog box, enter an appropriate name for your data source in the **Data Source Name** field. You use this name to establish a connection to your database. Here, enter `ORA` as the data source name. Enter a description for this data source, such as `Oracle database`, in the **Description** field. Enter your database name in the **TNS Service Name** field.



5. To establish the data source without Windows authentication, enter your user name in the **User ID** field. Or, to establish the data source with Windows authentication, leave this field blank. Leave **Application**, **Oracle**, **Workarounds**, and **SQLServer Migration** tabs with default settings.
6. 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.
7. Your database name and user name are automatically entered in the **Service Name** and **User Name** fields. Enter your password in the **Password** field. Click **OK**. If your computer successfully connects to the database, this message appears in the Testing Connection dialog box: Connection successful. Click **OK**.
8. 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 command line with the native ODBC connection.

Step 3. Connect using the native ODBC connection command line.

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

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

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

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

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database`

Oracle JDBC for Windows

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

In this section...

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.

Set up operating system authentication for Windows. Operating system authentication allows you to connect to your database using your system or network user name and password. In this case, the database does not require a different user name and password. Operating system authentication facilitates connection to the database and provides easy maintenance of database access credentials.

1. Ensure you have the latest Oracle OCI libraries installed on your computer. To install the latest library, see Driver Installation.
2. Run the `prefdir` command in the Command Window. The output of this command is a file path to a folder on your computer.
3. Close MATLAB if it is running.
4. Navigate to the folder and create a file called `javalibrarypath.txt` in the folder.
5. Open `javalibrarypath.txt` and insert the path to the Oracle OCI libraries. The entry should include the full path to the library files. The entry should not contain the library file names. For example, `C:\DB_Libraries\instantclient_11_2`.

6. Add the Oracle OCI library full path to the Windows Path environment variable.
7. Open MATLAB.

For details about Java libraries, see “Bring Java Classes into MATLAB Workspace”.

Step 3. Add the JDBC driver to the MATLAB static Java class path.

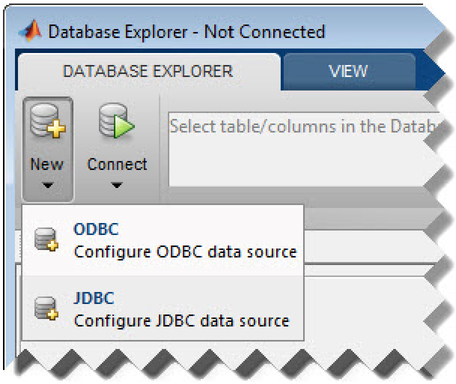
1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4. Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider and the JAR file name. For example, `C:\DB_Drivers\ojdbc6.jar`. Save and close `javaclasspath.txt`.
5. Restart MATLAB.

Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Bring Java Classes into MATLAB Workspace”.

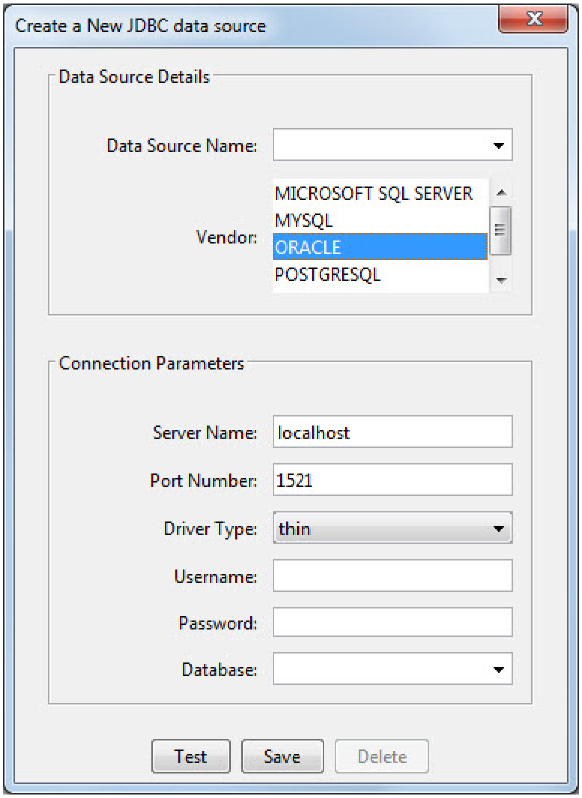
Step 4. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to Oracle using the JDBC connection command line.” on page 2-56

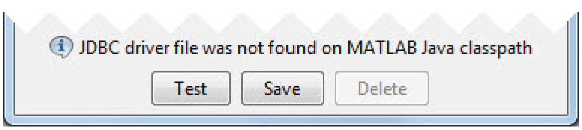
1. Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2. Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `ORACLE` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 3.



4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field.

5 To establish the data source with Windows authentication, set **Driver Type** to `oci`.

6. To establish the data source without Windows authentication, set **Driver Type** to `thin`.
7. Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!
8. Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.
9. If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

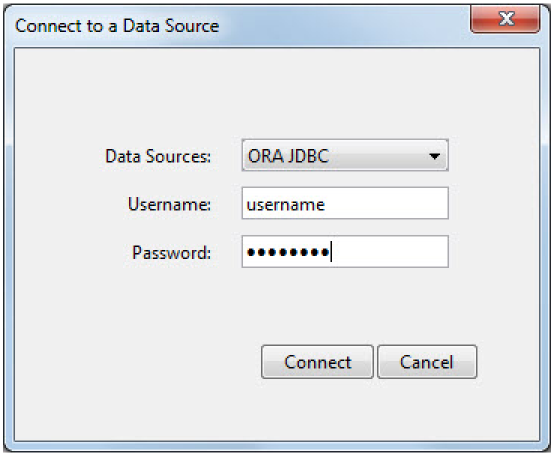
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

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

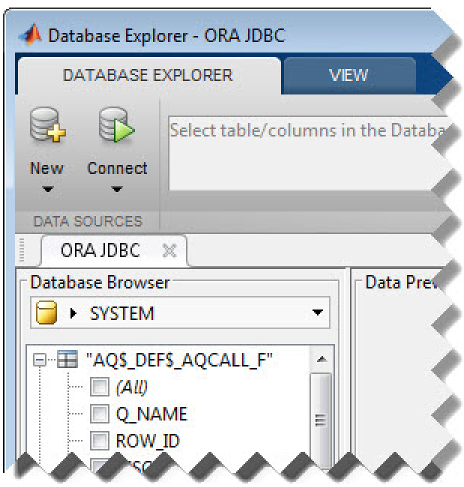
Connect to Oracle using Database Explorer.

1. After setting up the data source, to connect without Windows authentication, select the data source that you set up from the **Data Sources** list. Enter a user name and password. Click **Connect**.



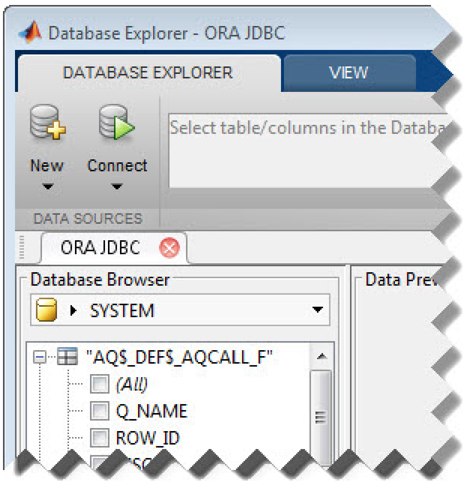
Or, to connect with Windows authentication, select the data source that you set up. Leave the user name and password blank. Click **Connect**.

Database Explorer connects to your database and displays its contents in a tab named with the data source name. You might need to select your database schema to display your database contents.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **ORA JDBC** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

   If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Oracle using the JDBC connection command line.

When using the command line, you do not have to set up a data source with Database Explorer. 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 `database` 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, the following code assumes you are connecting to a database named `dbname`, database server named `sname`, and port number `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 where the database or the Oracle client is installed.

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

Or, 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. For example, the following code assumes you are connecting to a database named dbname with user name username and password pwd.

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

If you have trouble using the database function to connect to your Oracle database, try using the full entry in your tnsnames.ora file in the URL string as one consecutive line. Leave the first argument blank. For example, the following code assumes the value of the URL name-value pair argument is set to the following 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 conn.

```
close(conn)
```

See Also

close | database | javaaddpath

More About

- “Working with Database Explorer” on page 4-2

- “Bring Java Classes into MATLAB Workspace”

MySQL ODBC for Windows

This tutorial shows how to set up a data source and connect to your MySQL database. This tutorial uses the MySQL ODBC 5.2a Driver to connect to the MySQL database.

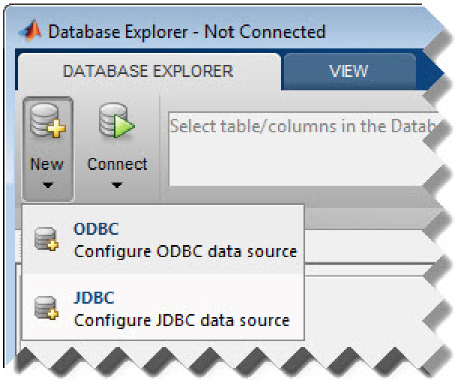
Step 1. Verify the driver installation.

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

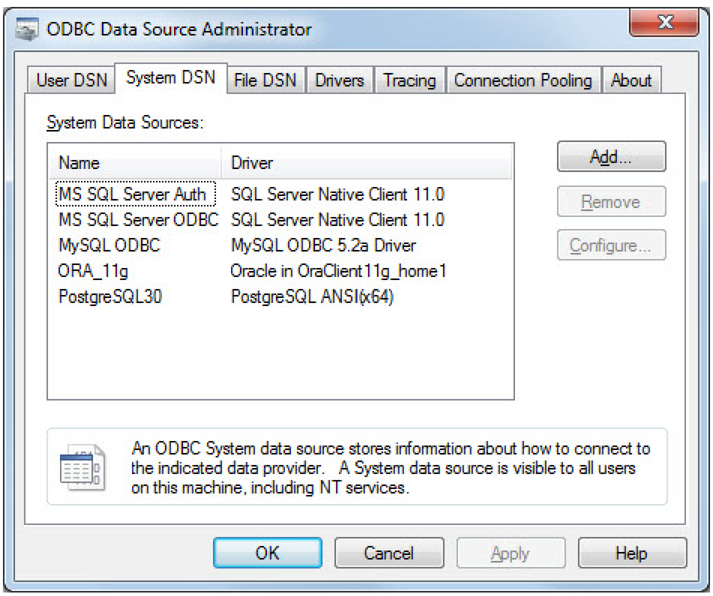
Note: The Database Toolbox no longer supports connection to a database using a 32-bit driver. Use a 64-bit version of MySQL. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see "MySQL JDBC for Windows" on page 2-65. For details about working with a 64-bit version of Windows, see http://www.mathworks.com/products/matlab/preparing-for-64-bit-windows.html.

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

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New** > **ODBC**.



The ODBC Data Source Administrator dialog box to define the ODBC data source opens.

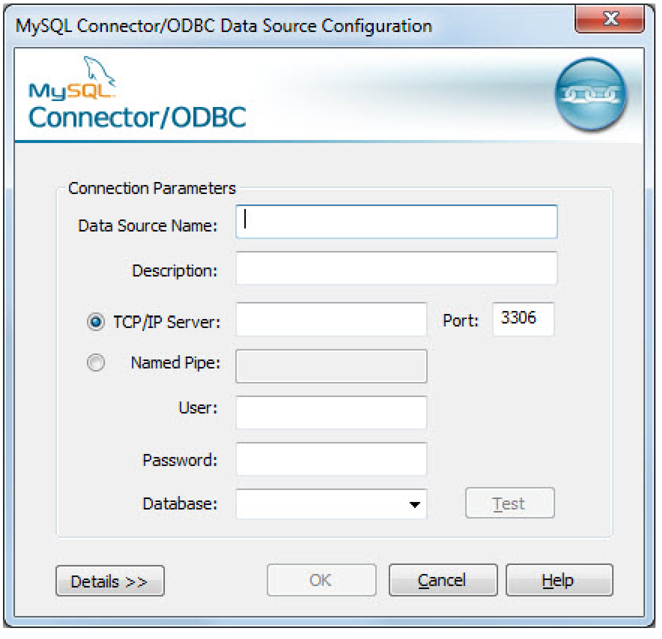


3 Click the **System DSN** tab and then click **Add**. When setting up an ODBC data source, you can use a User DSN or System DSN. A User DSN is specific to the user on a machine. Any data sources a user defines under User DSN are seen only by that specific user. Conversely, a System DSN is not specific to the user on a machine. Any

data sources a user defines under System DSN on a machine can be seen by any user who logs into that machine. Your ability to set up a User DSN or System DSN might depend on the database and ODBC driver you are using. For details, contact your database administrator or your database ODBC driver documentation.

4 A list of installed ODBC drivers appears in the Create New Data Source dialog box. Select the ODBC driver `MySQL ODBC 5.2a Driver`. Your ODBC driver might have a different name. Click **Finish**.

5 In the MySQL Connector/ODBC Data Source Configuration dialog box, enter an appropriate name for your data source in the **Data Source Name** field. You use this name to establish a connection to your database. Here, enter `MySQL` as the data source name. Enter a description for this data source, such as `MySQL database`, in the **Description** field. Enter your database server name in the **TCP/IP Server** field. Enter your port number in the **Port** field. The default port number is `3306`. Enter your user name in the **User** field. Enter your password in the **Password** field. Enter your database name in the **Database** field. Leave all tabs under the **Details** button with default settings.



6 Click **Test** to test the connection to your database. If your computer successfully connects to the database, the Test Result dialog box displays this message: Connection 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`.

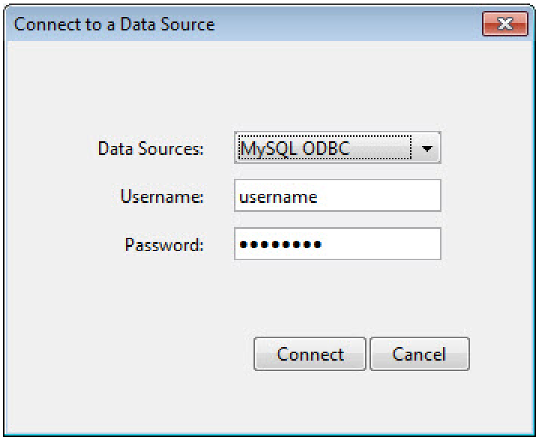
After you complete the data source setup, connect to the MySQL database using Database Explorer or the command line using the native ODBC connection.

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

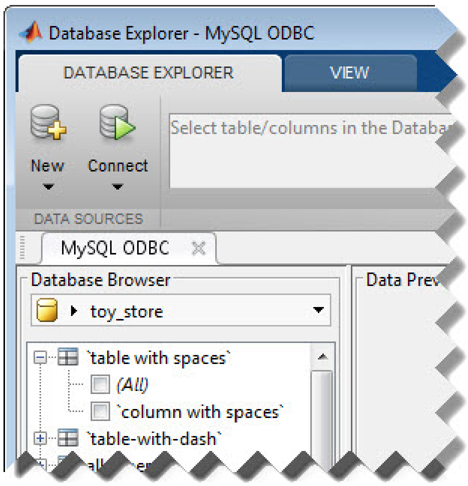
Connect to MySQL using Database Explorer.

1 After setting up the data source, click **Connect** in the **Database Explorer** tab.

2 In the Connect to a Data Source dialog box, connect to your database by selecting the data source name for the MySQL database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

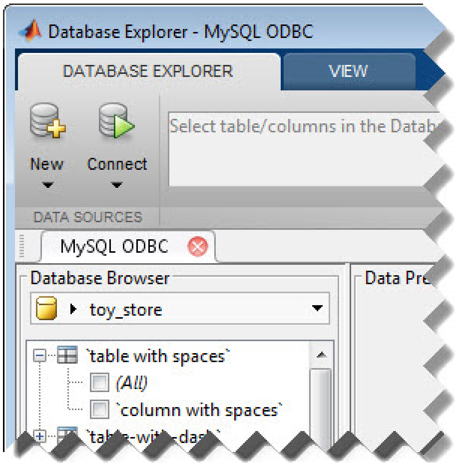


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



3 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **MySQL ODBC** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to MySQL using the native ODBC connection command line.

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

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

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database`

More About

- "Working with Database Explorer" on page 4-2

MySQL JDBC for Windows

This tutorial shows how to set up a data source and connect to your MySQL database. This tutorial uses the MySQL Connector/J 5.1.17 driver to connect to the MySQL Version 5.5.16 database.

In this section...

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.

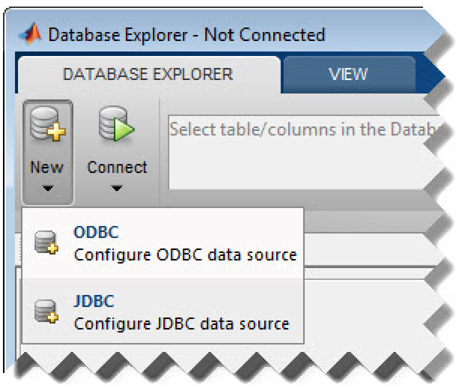
1. Run the `prefdir` command in the Command Window. The output of this command is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4. Open `javaclasspath.txt`. Add the full path to the database 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 and the JAR file name. For example, `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 your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Bring Java Classes into MATLAB Workspace”.

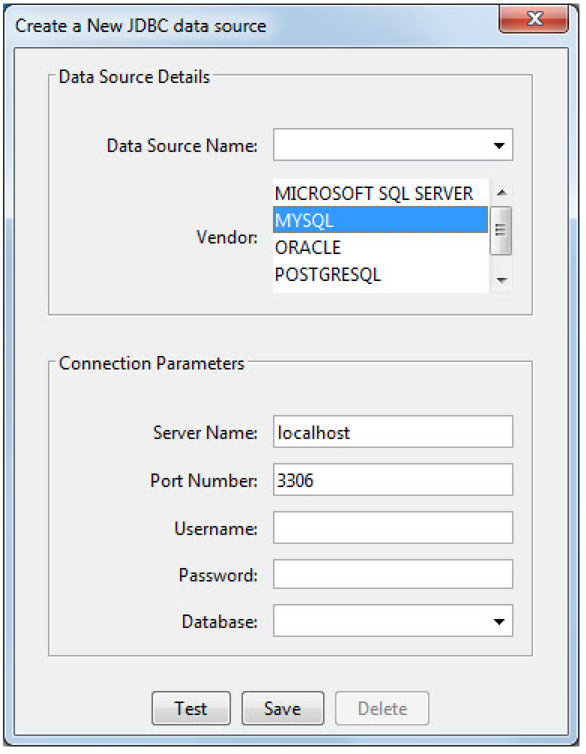
Step 3. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to MySQL using the JDBC connection command line.” on page 2-70

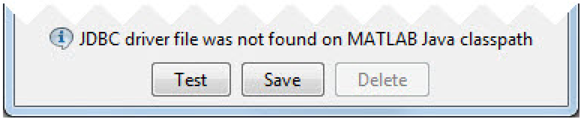
1. Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2. Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `MYSQL` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.



4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field.

5 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

6 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

7 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

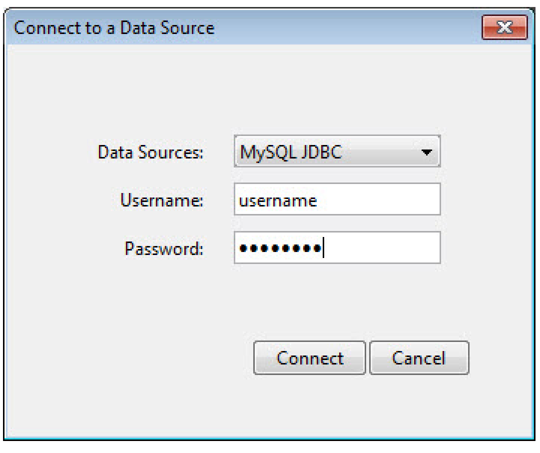
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

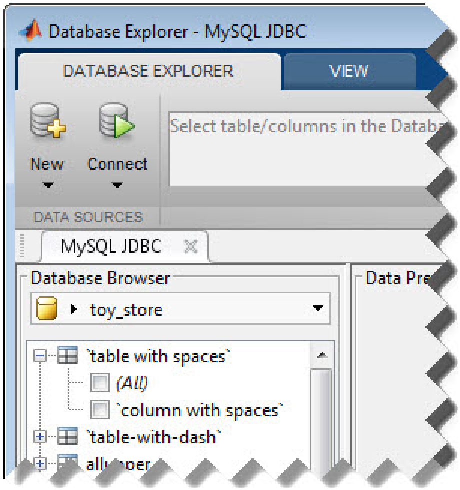
Step 4. Connect using Database Explorer or the command line.

Connect to MySQL using Database Explorer.

1 After setting up the data source, connect to your database by selecting the data source name for the MySQL database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

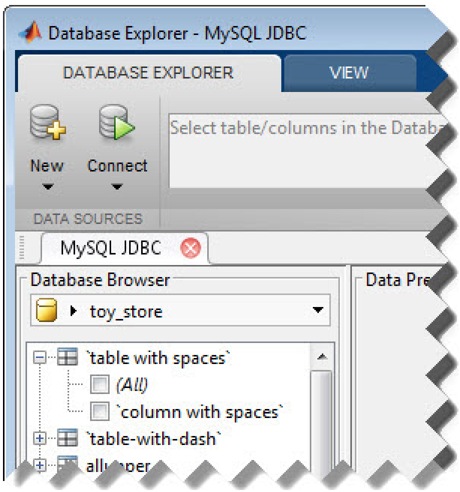


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **MySQL JDBC** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to MySQL using the JDBC connection command line.

1 Use the `Vendor` name-value pair argument of `database` to specify a connection to a MySQL database. For example, the following code assumes 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');
```

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2
- "Bring Java Classes into MATLAB Workspace"

PostgreSQL ODBC for Windows

This tutorial shows how to set up a data source and connect to your PostgreSQL database. This tutorial uses the PostgreSQL ANSI(x64) driver to connect to the PostgreSQL 9.2 database.

In this section...

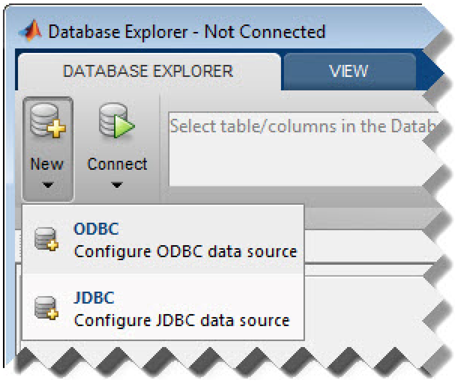
Step 1. Verify the driver installation.

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.

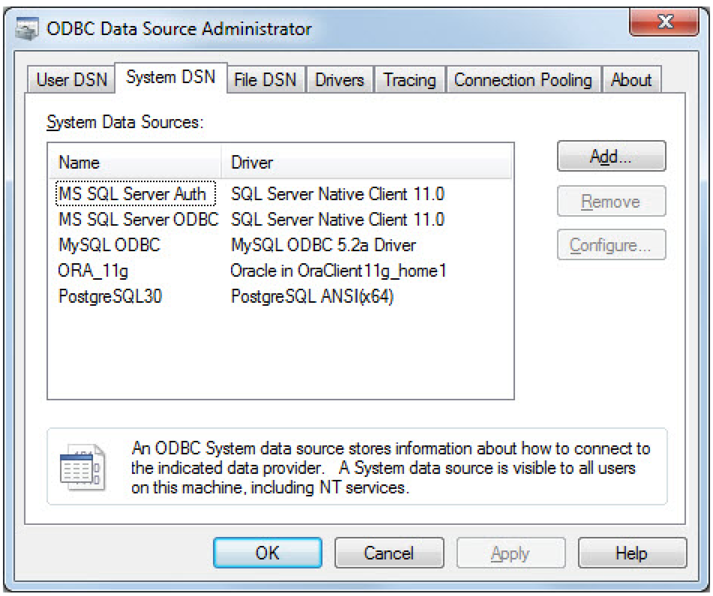
Note: The Database Toolbox no longer supports connection to a database using a 32-bit driver. Use a 64-bit version of PostgreSQL. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see "PostgreSQL JDBC for Windows" on page 2-77. For details about working with a 64-bit version of Windows, see http://www.mathworks.com/products/matlab/preparing-for-64-bit-windows.html.

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

1. Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2. Click the **Database Explorer** tab, and then select **New** > **ODBC**.



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

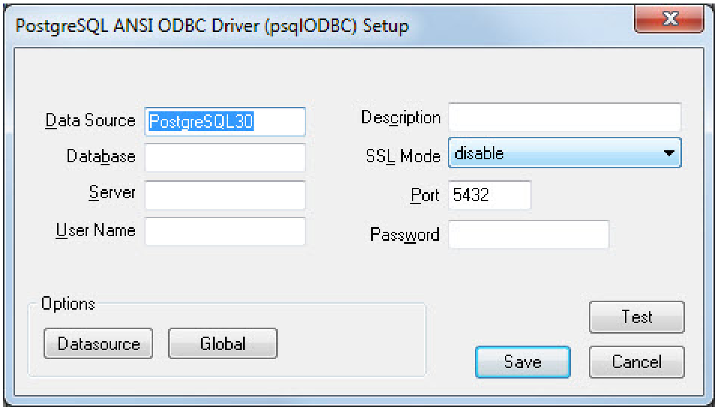


3 Click the **System DSN** tab and then click **Add**. When setting up an ODBC data source, you can use a User DSN or System DSN. A User DSN is specific to the user on a machine. Any data sources a user defines under User DSN are seen only by that specific user. Conversely, a System DSN is not specific to the user on a machine. Any

data sources a user defines under System DSN on a machine can be seen by any user who logs into that machine. Your ability to set up a User DSN or System DSN might depend on the database and ODBC driver you are using. For details, contact your database administrator or your database ODBC driver documentation.

4 A list of installed ODBC drivers appears in the Create New Data Source dialog box. Select the ODBC driver `PostgreSQL ANSI(x64)`. Your ODBC driver might have a different name. Click **Finish**.

5 In the PostgreSQL ANSI ODBC Driver (psqlODBC) Setup dialog box, enter an appropriate name for your data source in the **Data Source** field. You use this name to establish a connection to your database. Here, enter `PostgreSQL30` as the data source name. Enter a description for this data source, such as `PostgreSQL database`, in the **Description** field. Enter your database name in the **Database** field. Enter your database server name in the **Server** field. Enter your port number in the **Port** field. The default port number is `5432`. Enter your user name in the **User Name** field. Enter your password in the **Password** field. Leave all settings in the **Options** section with default settings.



6 Click **Test** to test the connection to your database. If your computer successfully connects to the database, the Connection Test dialog box displays this message: Connection 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`.

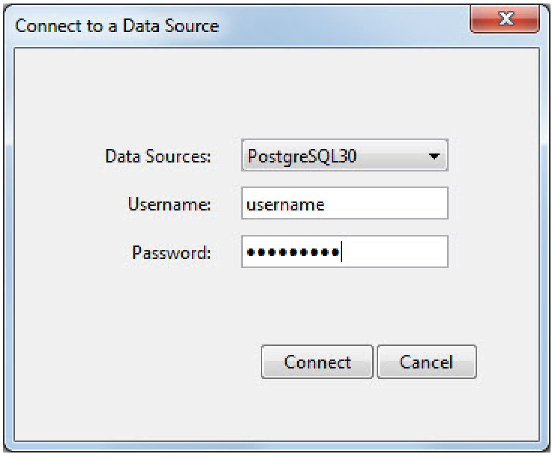
After you complete the data source setup, connect to the PostgreSQL database using Database Explorer or the native ODBC connection command line.

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

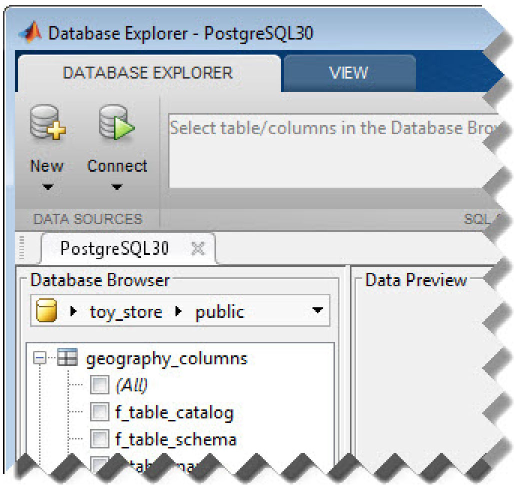
Connect to PostgreSQL using Database Explorer.

If you experience issues connecting using Database Explorer, use the command line with the native ODBC interface or JDBC to connect to your database.

1 After setting up the data source, click **Connect** in the **Database Explorer** tab.
2 In the Connect to a Data Source dialog box, connect to your database by selecting the data source name for the PostgreSQL database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

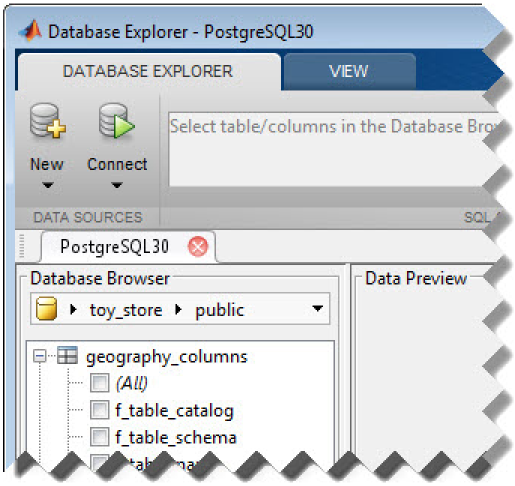


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



3 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **PostgreSQL30** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to PostgreSQL using the native ODBC connection command line.

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

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

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database`

More About

- "Working with Database Explorer" on page 4-2

PostgreSQL JDBC for Windows

This tutorial shows how to set up a data source and connect to your PostgreSQL database. This tutorial uses the JDBC4 PostgreSQL Driver, Version 8.4 to connect to the PostgreSQL 9.2 database.

In this section...

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` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `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 and the JAR file name. For example, `C:\DB_Drivers\postgresql-8.4-702.jdbc4.jar`. Save and close `javaclasspath.txt`.
5. Restart MATLAB.

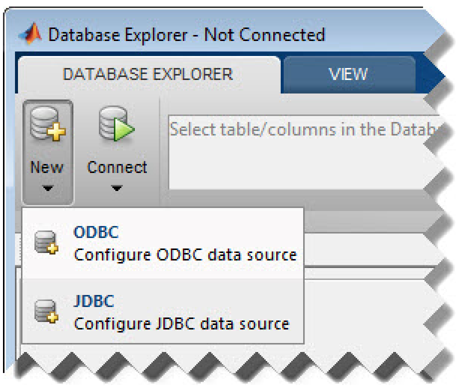
Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Bring Java Classes into MATLAB Workspace”.

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

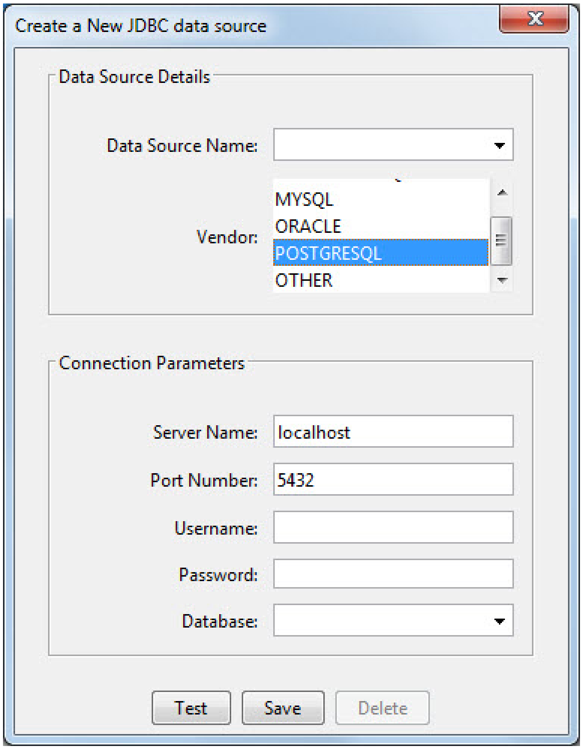
This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see "Connect to PostgreSQL using the JDBC connection command line." on page 2-82

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.

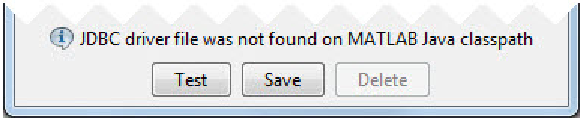
2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `POSTGRESQL` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.



4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field.

5 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

6 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

7 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

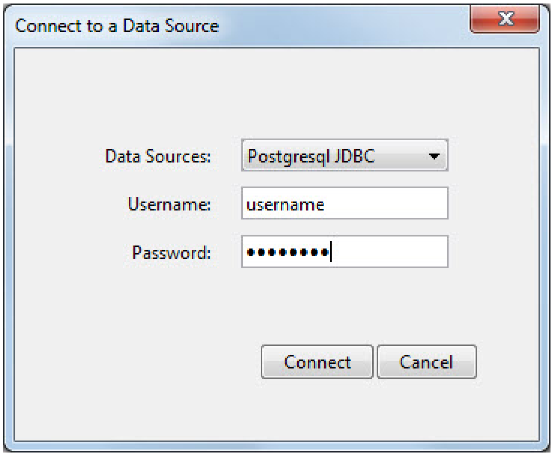
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

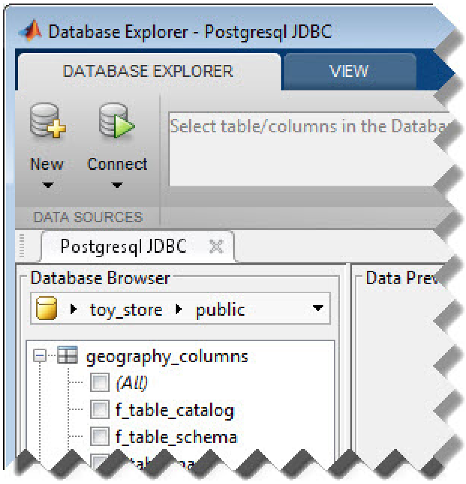
Step 4. Connect using Database Explorer or the command line.

Connect to PostgreSQL using Database Explorer.

1 After setting up the data source, connect to your database by selecting the data source name for the PostgreSQL database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

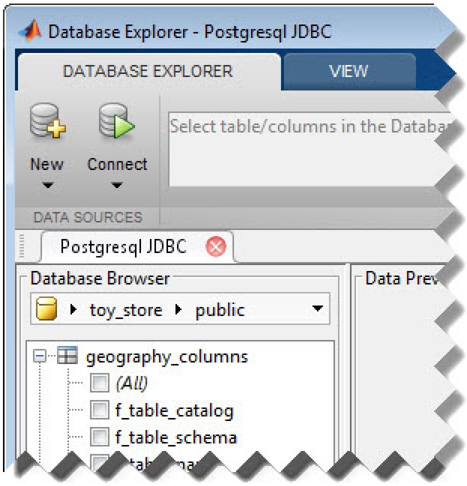


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **Postgresql JDBC** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to PostgreSQL using the JDBC connection command line.

1 Use the `Vendor` name-value pair argument of `database` to specify a connection to a PostgreSQL database. For example, the following code assumes 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','PostgreSQL',...
                'Server','sname');
```

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2
- "Bring Java Classes into MATLAB Workspace"

SQLite JDBC for Windows

This tutorial shows how to set up a data source and connect to your SQLite database. This tutorial uses the SQLite JDBC 3.7.2 Driver to connect to the SQLite Version 3.7.17 database.

Step 1. Verify the driver installation.

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

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 the MATLAB Interface to SQLite" on page 2-6.

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

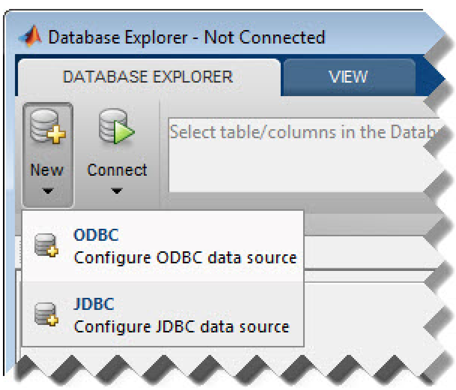
1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4. Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider and the JAR file name. For example, `C:\DB_Drivers\sqlite-jdbc-3.7.2.jar`. Save and close `javaclasspath.txt`.
5. Restart MATLAB.

Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Bring Java Classes into MATLAB Workspace”.

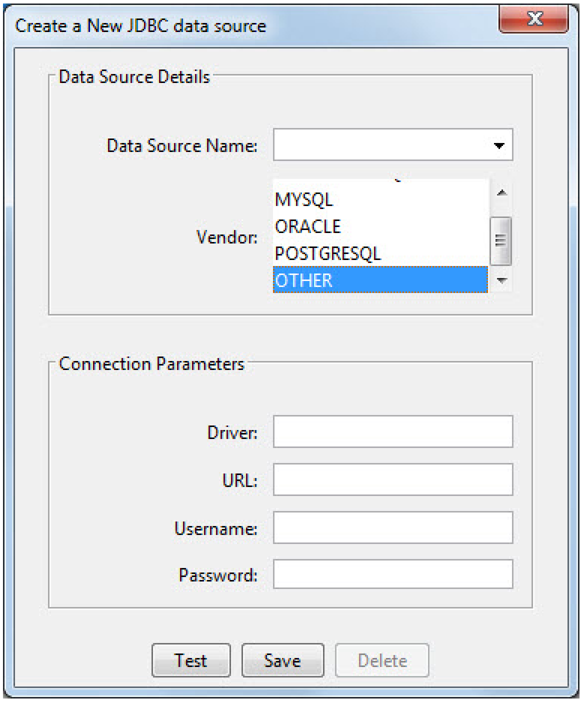
Step 3. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to SQLite using the JDBC connection command line.” on page 2-88 The driver and URL fields (in Database Explorer Create a New JDBC data source dialog box and in the `database` function) 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.

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New > JDBC**.

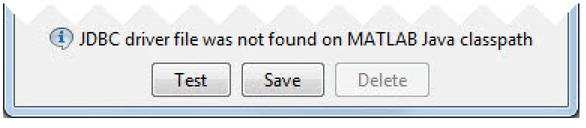


The Create a New JDBC data source dialog box opens.



3 Select `OTHER` from the **Vendor** list.

4 Enter the SQLite driver Java class object in the **Driver** field. Here, use `org.sqlite.JDBC`. After entering the driver, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.



5 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, `subprotocol` is `sqlite`. The last part of the URL string is `subname`. For SQLite, this 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 into the **URL** field.

6 Enter your user name in the **Username** field and your password in the **Password** field, or leave them blank if your database does not need them. Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

7 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

8 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

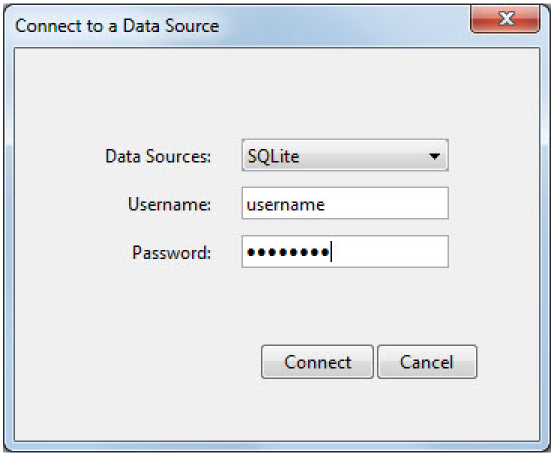
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

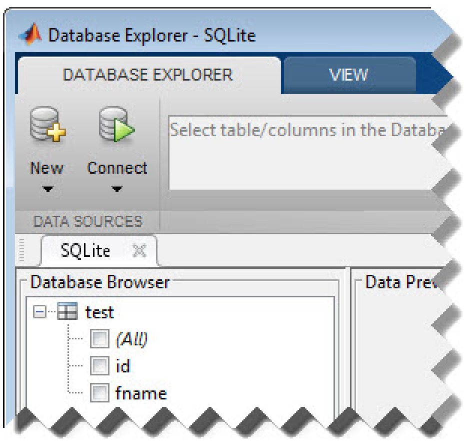
Step 4. Connect using Database Explorer or the command line.

Connect to SQLite using Database Explorer.

1 After setting up the data source, connect to your database by selecting the data source name for the SQLite database from the **Data Sources** list. Enter a user name and password or leave them blank if your database does not require them. Click **Connect**.

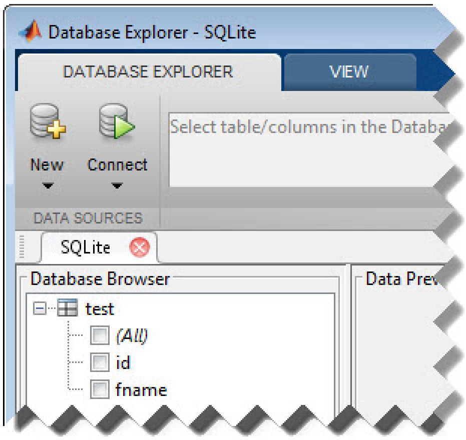


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **SQLite** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button (X) to close all database connections and Database Explorer.



Connect to SQLite using the JDBC connection command line.

1 Create 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, `subprotocol` is `sqlite`. The last part of the URL string is `subname`. For SQLite, this 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.

2 Connect to the SQLite database by using the `database` function. Enter the full path to your SQLite database `dbpath` for the first argument, or leave this argument blank and include the full path in the URL string `URL`. Enter your user name `username` and password `pwd`, or leave these blank if your database does not require them. The fourth argument is the driver Java class object. This code assumes the class object is `org.sqlite.JDBC`. The last argument is the URL string `URL`.

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

3 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2
- "Bring Java Classes into MATLAB Workspace"

Sybase ODBC for Windows

This tutorial shows how to set up a data source and connect to your Sybase database. This tutorial uses the Adaptive Server Enterprise ODBC Driver to connect to the Sybase Adaptive Server Enterprise 15.7 database.

In this section...

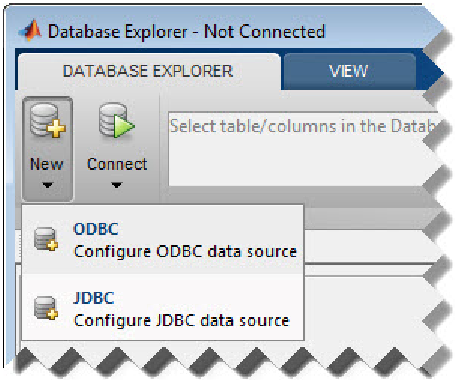
Step 1. Verify the driver installation.

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.

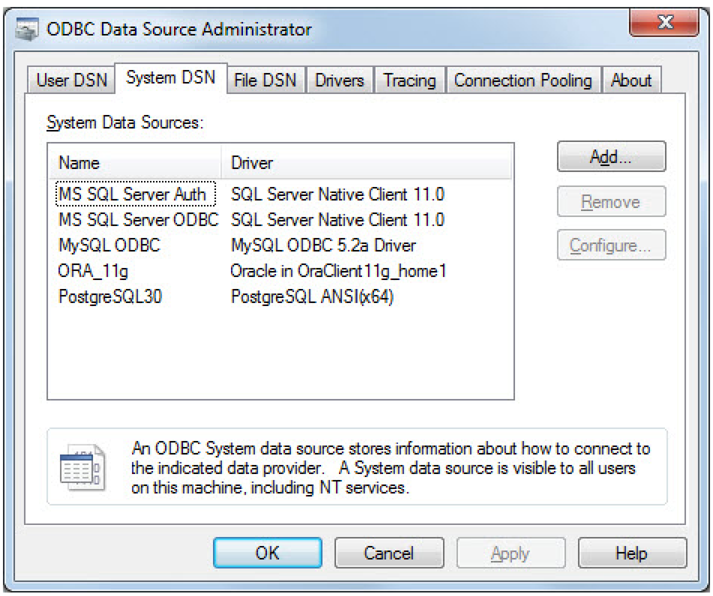
Note: The Database Toolbox no longer supports connection to a database using a 32-bit driver. Use a 64-bit version of Sybase. If you have issues working with the ODBC driver, use the JDBC driver instead. For details, see "Sybase JDBC for Windows" on page 2-97. For details about working with a 64-bit version of Windows, see http://www.mathworks.com/products/matlab/preparing-for-64-bit-windows.html.

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

1. Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2. Click the **Database Explorer** tab, and then select **New** > **ODBC**.



The ODBC Data Source Administrator dialog box to define the ODBC data source opens.

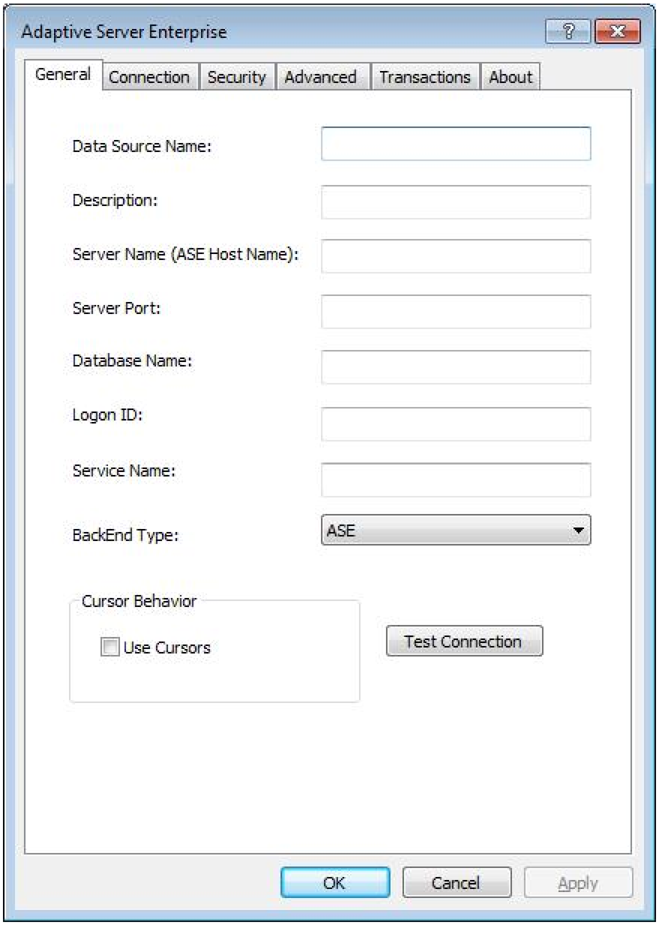


3 Click the **System DSN** tab and then click **Add**. When setting up an ODBC data source, you can use a User DSN or System DSN. A User DSN is specific to the user on a machine. Any data sources a user defines under User DSN are seen only by that specific user. Conversely, a System DSN is not specific to the user on a machine. Any

data sources a user defines under System DSN on a machine can be seen by any user who logs into that machine. Your ability to set up a User DSN or System DSN might depend on the database and ODBC driver you are using. For details, contact your database administrator or your database ODBC driver documentation.

4 A list of installed ODBC drivers appears in the Create New Data Source dialog box. Select the ODBC driver `Adaptive Server Enterprise`. Your ODBC driver might have a different name. Click **Finish**.

5 In the Adaptive Server Enterprise dialog box, enter an appropriate name for your data source in the **Data Source Name** field. You use this name to establish a connection to your database. Here, enter `Sybase` as the data source name. Enter a description for this data source, such as `Sybase database`, in the **Description** field. Enter your database server name in the **Server Name (ASE Host Name)** field. Enter your port number in the **Server Port** field. Enter your database name in the **Database Name** field. Enter your user name in the **Logon ID** field. Leave all other tabs with default settings.



6 Click **Test Connection** to test the connection to your database. Another screen appears with login information. Enter your user name in the **Logon ID** field and your password in the **Password** field. The other three fields are prepopulated with your specific data.

7 Click **OK**. If your computer successfully connects to the database, the dialog box displays Login Succeeded.

8 Click **OK** in the Adaptive Server Enterprise dialog boxes to close them. The ODBC Data Source Administrator dialog box shows the ODBC data source `Sybase`.

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

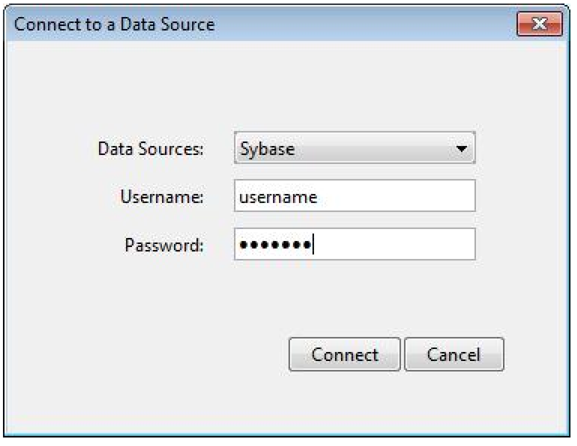
Step 3. Connect using Database Explorer or the command line.

Connect to Sybase using Database Explorer.

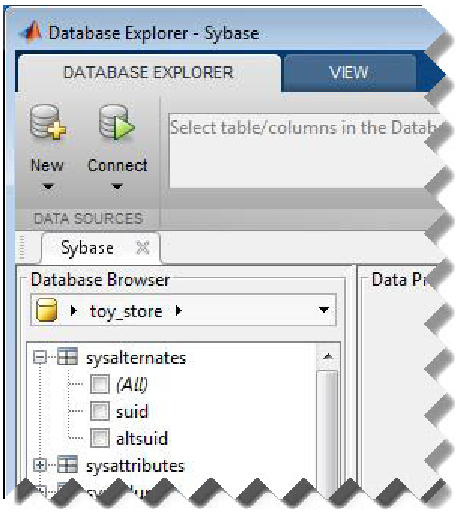
If you experience issues connecting using Database Explorer, use the native ODBC interface with the command line or JDBC to connect to your database.

1 After setting up the data source, click **Connect** in the **Database Explorer** tab.

2 In the Connect to a Data Source dialog box, connect to your database by selecting the data source name for the Sybase database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

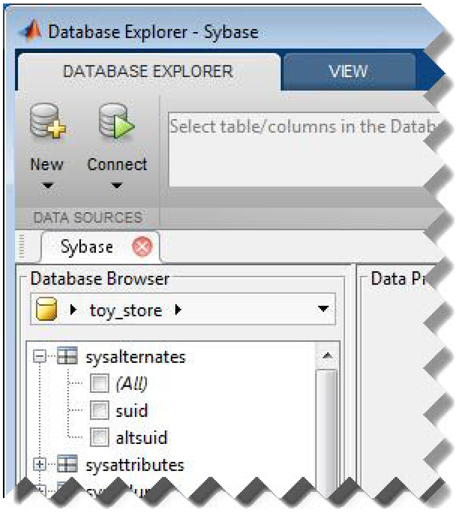


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



3 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **Sybase** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Sybase using the native ODBC connection command line.

1 Connect to your database with the ODBC data source name. For example, the following code assumes you are connecting to a data source named `Sybase` with user name `username` and password `pwd`.

```
conn = database.ODBCConnection('Sybase','username','pwd');
```

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database`

More About

- "Working with Database Explorer" on page 4-2

Sybase JDBC for Windows

This tutorial shows how to set up a data source and connect to your Sybase database. This tutorial uses the jConnect 4 JDBC Driver to connect to the Sybase Adaptive Server Enterprise 15.7 database.

In this section...

Step 1. Verify the driver installation.

If the JDBC driver for Sybase 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` command in the Command Window. The output is a file path to a folder on your computer.
2 Close MATLAB if it is running.
3 Navigate to the folder and create a file called `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 and the JAR file name. For example, `C:\DB_Drivers\jconn4.jar`. Save and close `javaclasspath.txt`.
5 Restart MATLAB.

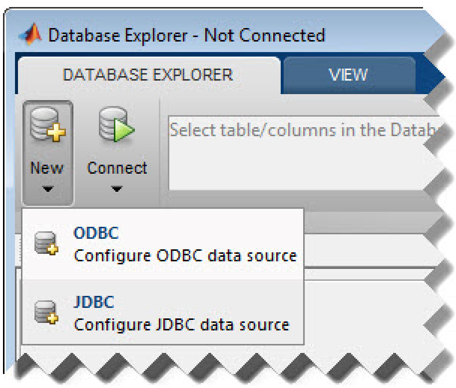
Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Bring Java Classes into MATLAB Workspace”.

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

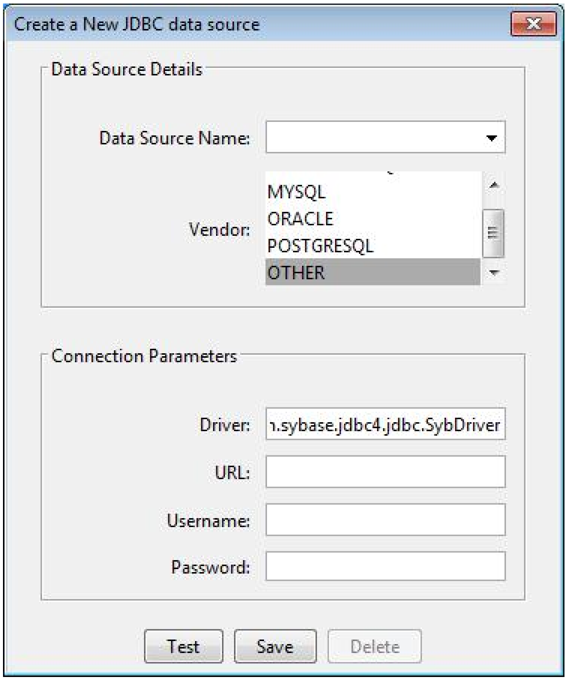
This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see "Connect to Sybase using the JDBC connection command line." on page 2-102 The driver and URL fields (in the Database Explorer Create a New JDBC data source dialog box and in the `database` function) 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.

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.

2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.

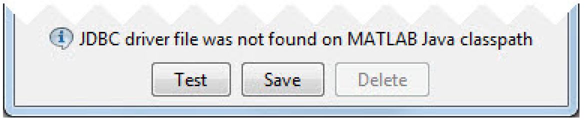


The Create a New JDBC data source dialog box opens.



3 Select `OTHER` from the **Vendor** list.

4 Enter the Sybase driver Java class object in the **Driver** field. Here, use `com.sybase.jdbc4.jdbc.SybDriver`. After entering the driver, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.



5 Connect to the Sybase 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, `subprotocol` is `sybase:Tds`. The last part of the URL string is `subname`. For Sybase, this contains the server name, the port number, and the database name. For example, your string is `jdbc:sybase:Tds:ServerName:PortNumber/dbname`, where `ServerName` is your server name, `PortNumber` is your port number, and `dbname` is your database name. Enter your full string into the **URL** field.

6 Enter your user name in the **Username** field and your password in the **Password** field. Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

7 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

8 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

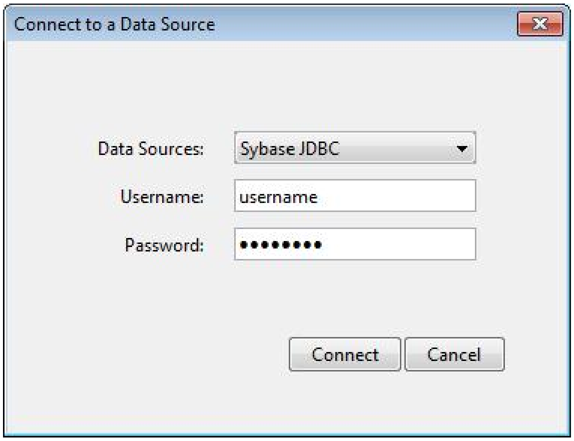
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

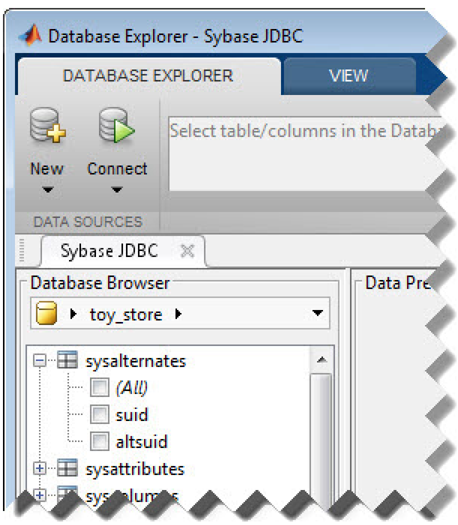
Step 4. Connect using Database Explorer or the command line.

Connect to Sybase using Database Explorer.

1 After setting up the data source, connect to your database by selecting the data source name for the Sybase database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

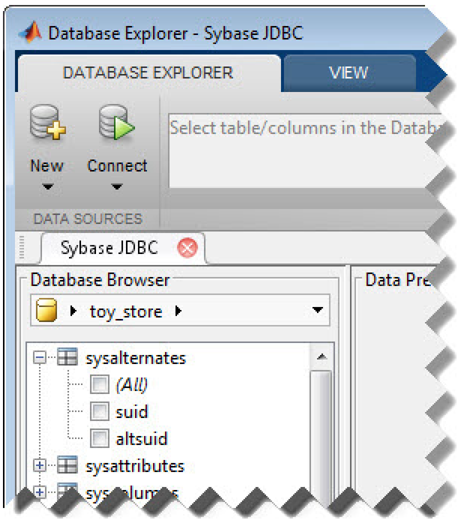


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **Sybase JDBC** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Sybase using the JDBC connection command line.

1. Create 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, `subprotocol` is `sybase:Tds`. The last part of the URL string is `subname`. For Sybase, this contains the server name, the port number, and the database name. For example, your URL string is `jdbc:sybase:Tds:ServerName:PortNumber/dbname`, where `ServerName` is your server name, `PortNumber` is your port number, and `dbname` is your database name.
2. Connect to the Sybase database using the `database` function. For example, the following code assumes you are connecting to a database named `dbname` with user name `username` and password `pwd`. The fourth argument is the driver Java class object. This code assumes the class object is `com.sybase.jdbc4.jdbc.SybDriver`. The last argument is the URL string `URL`.

```
conn = database('dbname','username','pwd',...
```

```
               'com.sybase.jdbc4.jdbc.SybDriver','URL');
```

3 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2
- "Bring Java Classes into MATLAB Workspace"

Microsoft SQL Server JDBC for Mac OS X

This tutorial shows how to set up a data source and connect to your Microsoft SQL Server database. This tutorial uses the Microsoft JDBC Driver 4.0 for Microsoft SQL Server to connect to the Microsoft SQL Server 2012 Express database.

Step 1. Verify the driver installation.

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

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

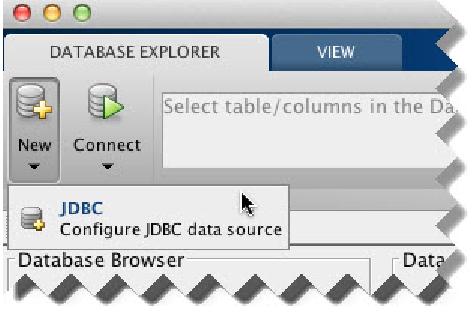
1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4. Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider and the JAR file name. For example, `/home/user/DB_Drivers/sqljdbc4.jar`. Save and close `javaclasspath.txt`.
5. Restart MATLAB.

Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".

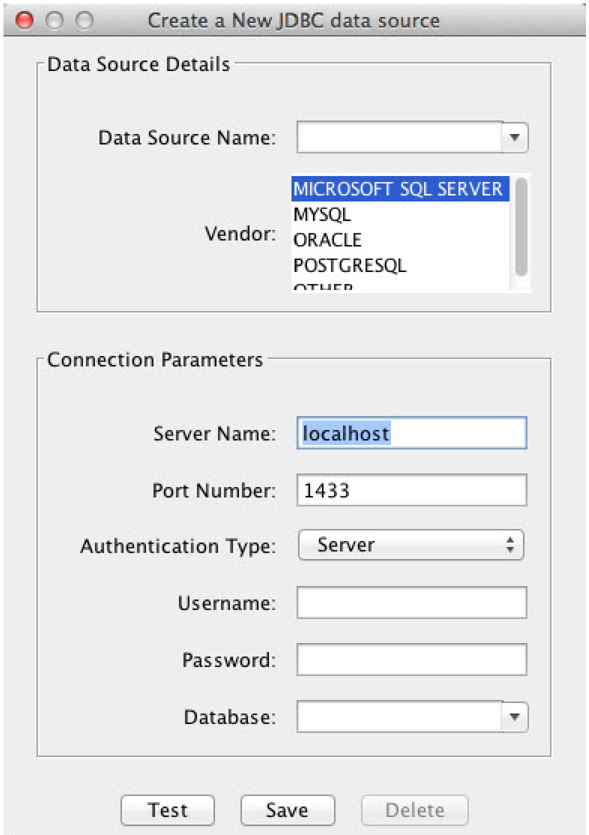
Step 3. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to Microsoft SQL Server using the JDBC connection command line.” on page 2-109

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `MICROSOFT SQL SERVER` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

4 Enter the database server name in the **Server Name**, port number in the **Port Number** field, user name in the **Username** field, password in the **Password**

field, and database name in the **Database** field. Set the **Authentication Type** to `Server`.

5 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

6 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

7 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

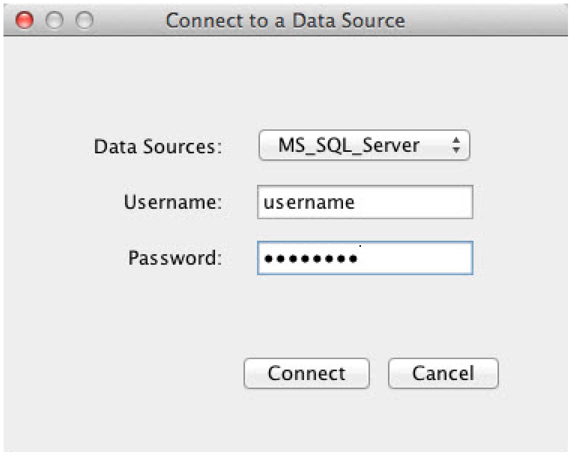
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

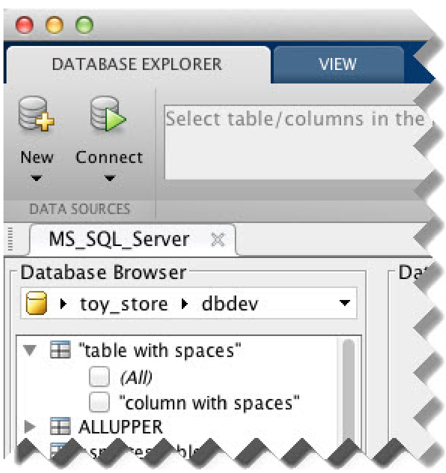
Step 4. Connect using Database Explorer or the command line.

Connect to Microsoft SQL Server using Database Explorer.

1 After setting up the data source, select the data source that you set up from the **Data Sources** list. Enter a user name and password. Click **Connect**.



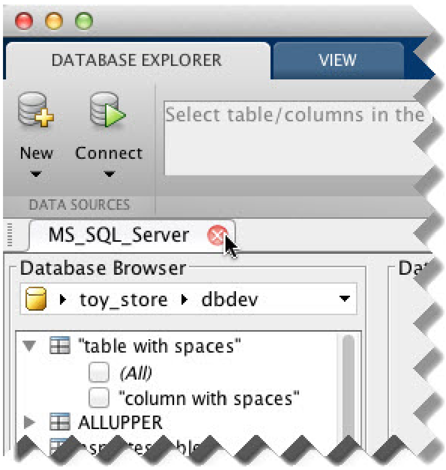
Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **MS_SQL_Server** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If

you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-left corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Microsoft SQL Server using the JDBC connection command line.

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

1 Use the `Vendor` name-value pair argument of `database` to specify a connection to a Microsoft SQL Server database. Set the `AuthType` name-value pair argument to `Server`. For example, the following code assumes you are connecting to a database named `dbname` on a database server named `sname` with user name `username`, password `pwd`, and port number as `123456`.

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

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- “Working with Database Explorer” on page 4-2
- “Bring Java Classes into MATLAB Workspace”

Microsoft SQL Server JDBC for Linux

This tutorial shows how to set up a data source and connect to your Microsoft SQL Server database. This tutorial uses the Microsoft JDBC Driver 4.0 for Microsoft SQL Server to connect to the Microsoft SQL Server 2012 Express database.

In this section...

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.

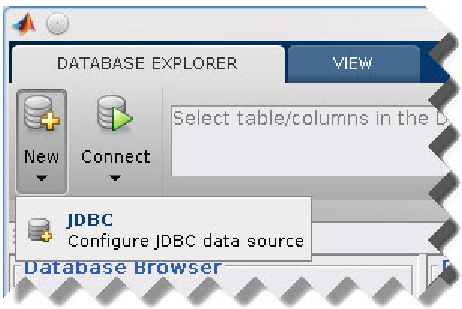
1 Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2 Close MATLAB if it is running.
3 Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4 Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider and the JAR file name. For example, `/home/user/DB_Drivers/sqljdbc4.jar`. Save and close `javaclasspath.txt`.
5 Restart MATLAB.

Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".

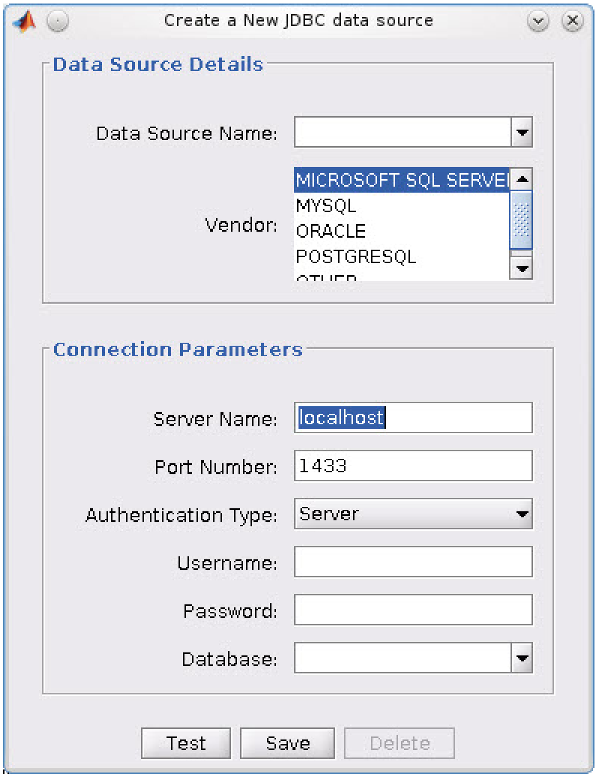
Step 3. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to Microsoft SQL Server using the JDBC connection command line.” on page 2-116

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `MICROSOFT SQL SERVER` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field. Set the **Authentication Type** to `Server`.

5 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

6 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

7 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

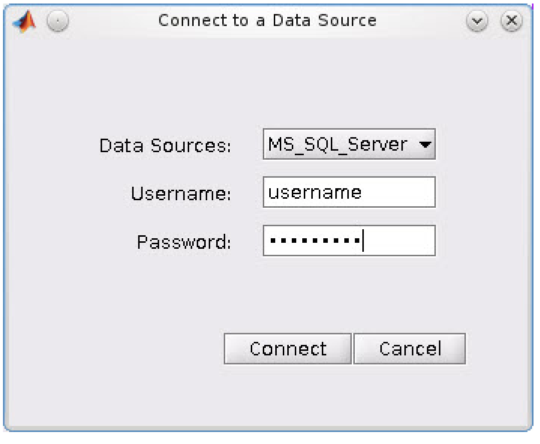
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

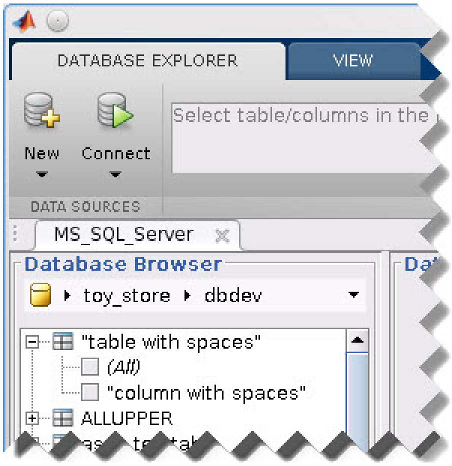
Step 4. Connect using Database Explorer or the command line.

Connect to Microsoft SQL Server using Database Explorer.

1 After setting up the data source, select the data source that you set up from the **Data Sources** list. Enter a user name and password. Click **Connect**.

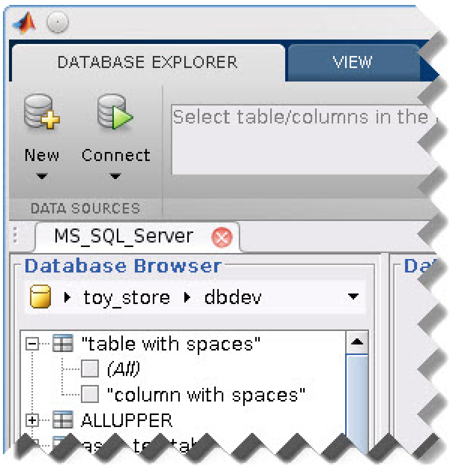


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **MS_SQL_Server** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Microsoft SQL Server using the JDBC connection command line.

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

1. Use the `Vendor` name-value pair argument of `database` to specify a connection to a Microsoft SQL Server database. Set the `AuthType` name-value pair argument to `Server`. For example, the following code assumes you are connecting to a database named `dbname` on a database server named `sname` with user name `username`, password `pwd`, and port number as `123456`.

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

2. Close the database connection `conn`.

   ```
   close(conn)
   ```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2

- “Bring Java Classes into MATLAB Workspace”

Oracle JDBC for Mac OS X

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

Step 1. Verify the driver installation.

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

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

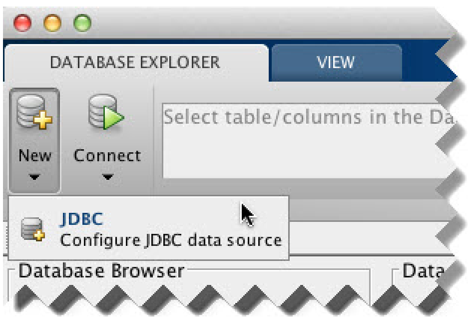
1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4. Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider and the JAR file name. For example, `/home/user/DB_Drivers/ojdbc6.jar`. Save and close `javaclasspath.txt`.
5. Restart MATLAB.

Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".

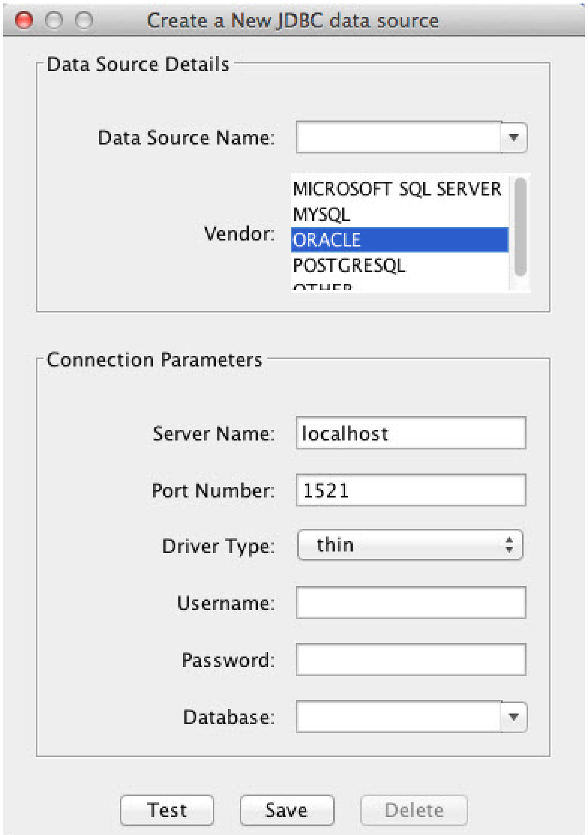
Step 3. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to Oracle using the JDBC connection command line.” on page 2-123

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `ORACLE` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field. Select **Driver Type** of `thin` or `oci`. Use `thin` as the default driver. Use `oci` if you installed an OCI driver.

5. Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!
6. Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.
7. If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

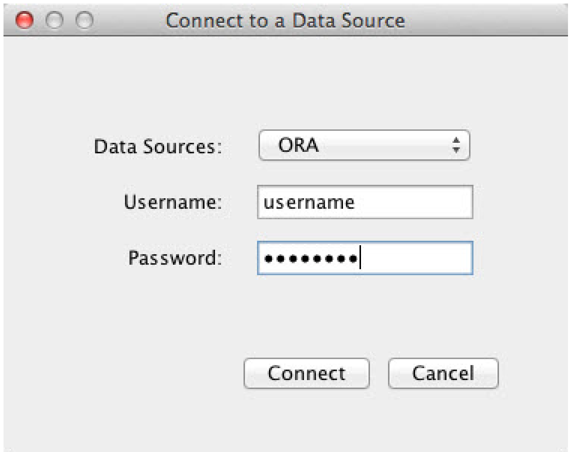
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

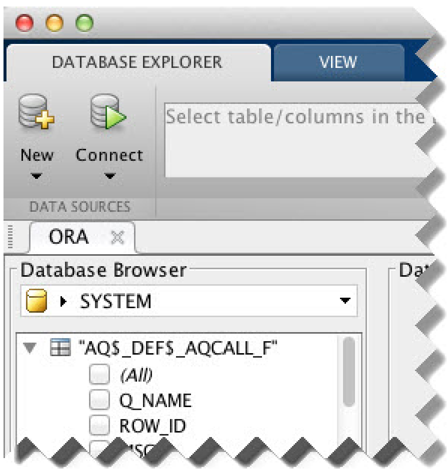
Step 4. Connect using Database Explorer or the command line.

Connect to Oracle using Database Explorer.

1. After setting up the data source, select the data source that you set up from the **Data Sources** list. Enter a user name and password. Click **Connect**.

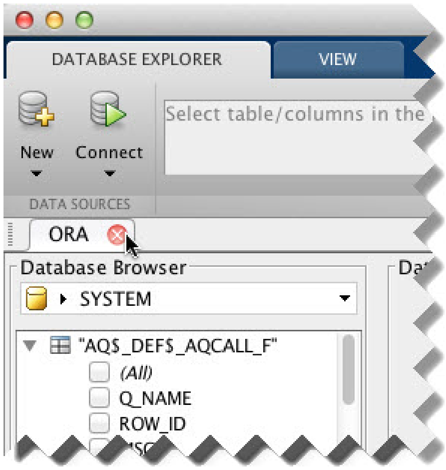


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button () next to the **ORA** data source name on the database tab. The **Close** button turns into a red circle (). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-left corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Oracle using the JDBC connection command line.

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

1 Use the `Vendor` name-value pair argument of `database` to specify a connection to an Oracle database. Set the `DriverType` name-value pair argument to `thin`. For example, the following code assumes you are connecting to a database named `dbname` on a database server named `sname` with user name `username`, password `pwd`, and port number as `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 where the database or the Oracle client is installed.

```
conn = database('dbname','username','pwd',...
                'Vendor','Oracle','DriverType','thin',...
```

```
                 'Server','sname','PortNumber',123456);
```

Or, if you have trouble using the `database` function to connect to your Oracle database, try using the full entry in your `tnsnames.ora` file in the URL string as one consecutive line. Leave the first argument blank. For example, the following code assumes the value of the `URL` name-value pair argument is set to the following `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 `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2
- "Bring Java Classes into MATLAB Workspace"

Oracle JDBC for Linux

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

In this section...

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` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `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 and the JAR file name. For example, `/home/user/DB_Drivers/ojdbc6.jar`. Save and close `javaclasspath.txt`.
5. Restart MATLAB.

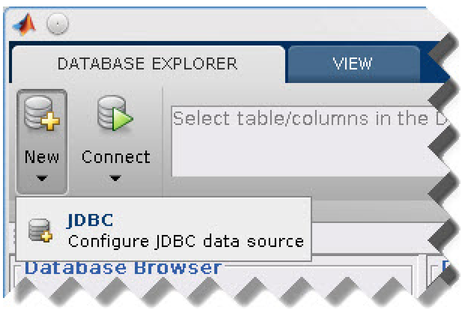
Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Bring Java Classes into MATLAB Workspace”.

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

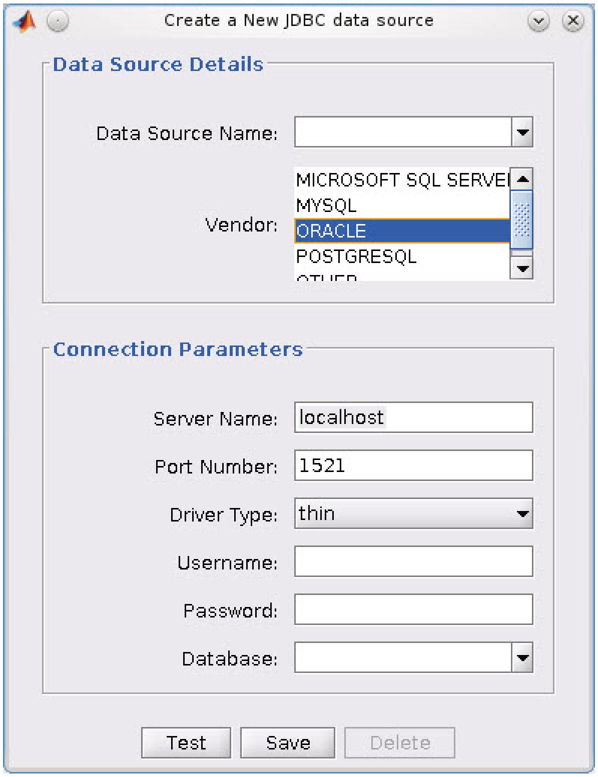
This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to Oracle using the JDBC connection command line.” on page 2-130

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.

2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `ORACLE` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field. Select **Driver Type** of `thin` or `oci`. Use `thin` as the default driver. Use `oci` if you installed an OCI driver.

5 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

6 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

7 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

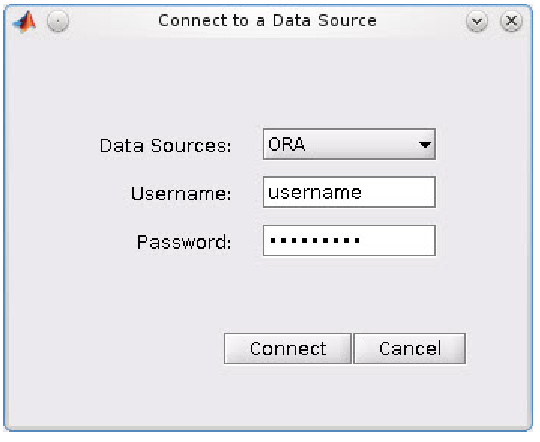
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

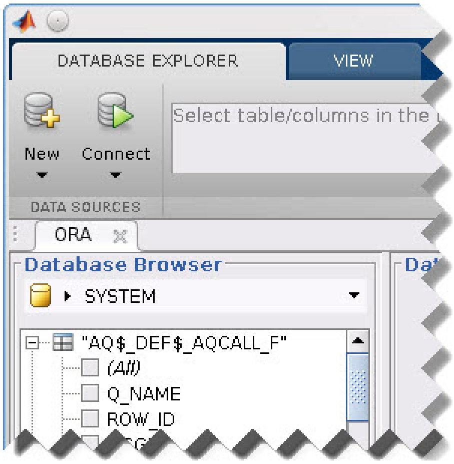
Step 4. Connect using Database Explorer or the command line.

Connect to Oracle using Database Explorer.

1 After setting up the data source, select the data source that you set up from the **Data Sources** list. Enter a user name and password. Click **Connect**.

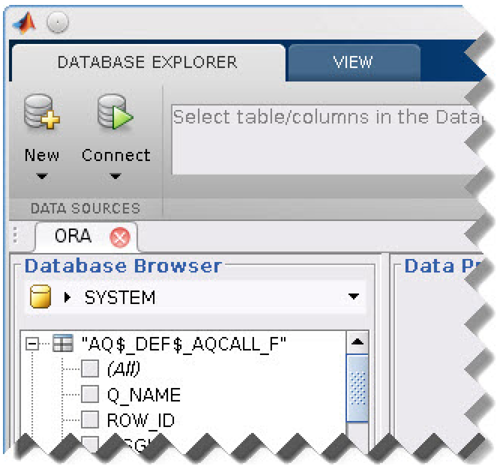


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **ORA** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Oracle using the JDBC connection command line.

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

1 Use the `Vendor` name-value pair argument of `database` to specify a connection to an Oracle database. Set the `DriverType` name-value pair argument to `thin`. For example, the following code assumes you are connecting to a database named `dbname` on a database server named `sname` with user name `username`, password `pwd`, and port number as `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 where the database or the Oracle client is installed.

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

   Or, if you have trouble using the `database` function to connect to your Oracle database, try using the full entry in your `tnsnames.ora` file in the URL string as one consecutive line. Leave the first argument blank. For example, the following code assumes the value of the `URL` name-value pair argument is set to the following `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 `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2
- "Bring Java Classes into MATLAB Workspace"

MySQL JDBC for Mac OS X

This tutorial shows how to set up a data source and connect to your MySQL database. This tutorial uses the MySQL Connector/J 5.1.17 driver to connect to the MySQL Version 5.5.16 database.

Step 1. Verify the driver installation.

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

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

1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `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 and the JAR file name. For example, `/home/user/DB_Drivers/mysql-connector-java-5.1.17-bin.jar`. Save and close `javaclasspath.txt`.
5. Restart MATLAB.

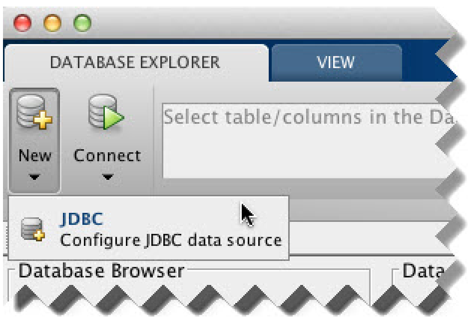
Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".

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

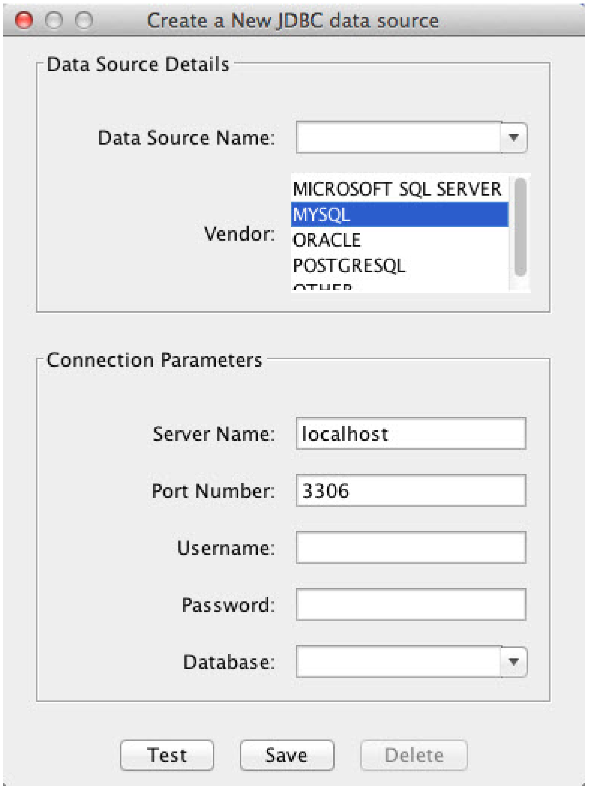
This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to MySQL using the JDBC connection command line.” on page 2-137

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.

2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `MYSQL` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field.

5 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

6 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

7 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

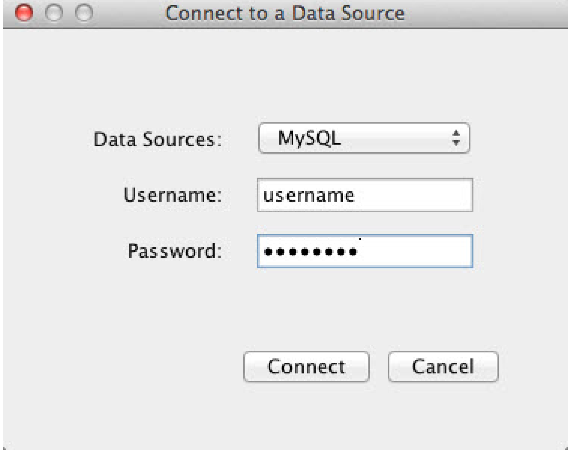
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

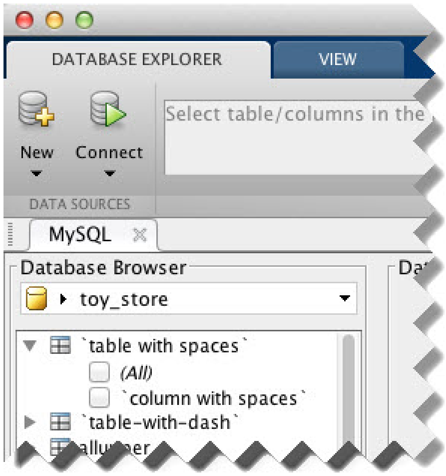
Step 4. Connect using Database Explorer or the command line.

Connect to MySQL using Database Explorer.

1 After setting up the data source, connect to your database by selecting the data source name for the MySQL database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

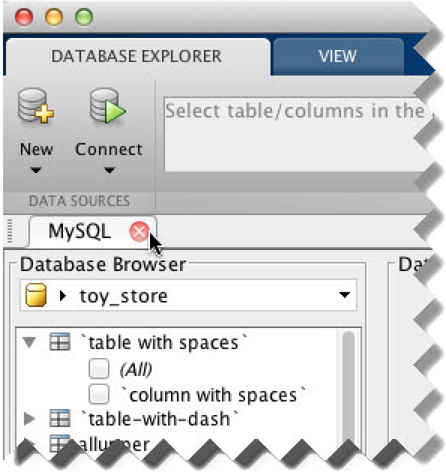


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button () next to the **MySQL** data source name on the database tab. The **Close** button turns into a red circle (). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button () in the top-left corner.

If Database Explorer is docked, click the **Close** button () to close all database connections and Database Explorer.



Connect to MySQL using the JDBC connection command line.

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

1. Use the `Vendor` name-value pair argument of `database` to specify a connection to a MySQL database. For example, the following code assumes 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');
```

2. Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- “Working with Database Explorer” on page 4-2

- “Bring Java Classes into MATLAB Workspace”

MySQL JDBC for Linux

This tutorial shows how to set up a data source and connect to your MySQL database. This tutorial uses the MySQL Connector/J 5.1.17 driver to connect to the MySQL Version 5.5.16 database.

Step 1. Verify the driver installation.

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

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

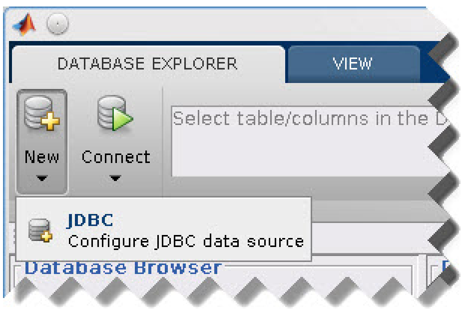
1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4. Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider and the JAR file name. For example, `/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 your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".

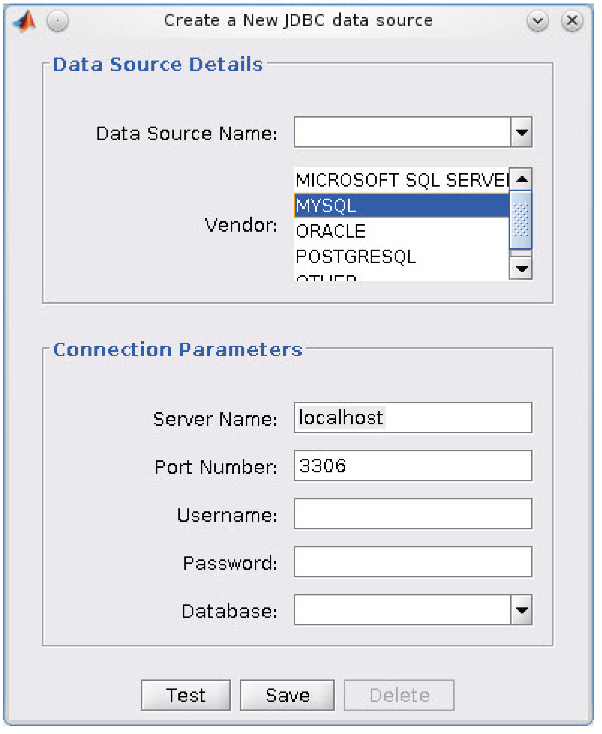
Step 3. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see "Connect to MySQL using the JDBC connection command line." on page 2-144

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `MYSQL` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field.

5 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

6 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

7 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

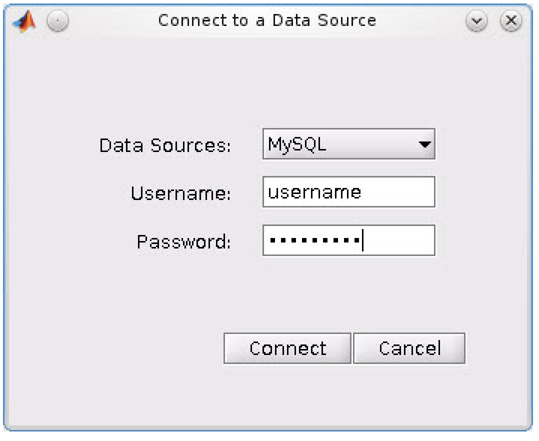
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

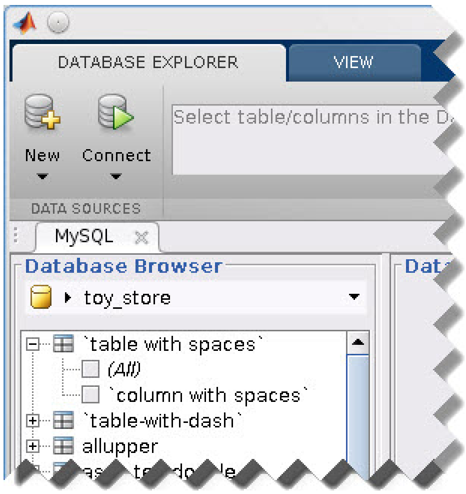
Step 4. Connect using Database Explorer or the command line.

Connect to MySQL using Database Explorer.

1 After setting up the data source, connect to your database by selecting the data source name for the MySQL database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

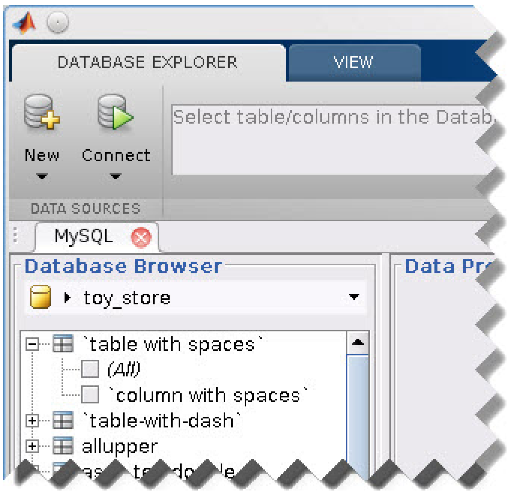


   Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **MySQL** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to MySQL using the JDBC connection command line.

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

1 Use the `Vendor` name-value pair argument of `database` to specify a connection to a MySQL database. For example, the following code assumes 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');
```

2 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2

- “Bring Java Classes into MATLAB Workspace”

PostgreSQL JDBC for Mac OS X

This tutorial shows how to set up a data source and connect to your PostgreSQL database. This tutorial uses the JDBC4 PostgreSQL Driver, Version 8.4 to connect to the PostgreSQL 9.2 database.

Step 1. Verify the driver installation.

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

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

1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `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 and the JAR file name. For example, `/home/user/DB_Drivers/postgresql-8.4-702.jdbc4.jar`. Save and close `javaclasspath.txt`.
5. Restart MATLAB.

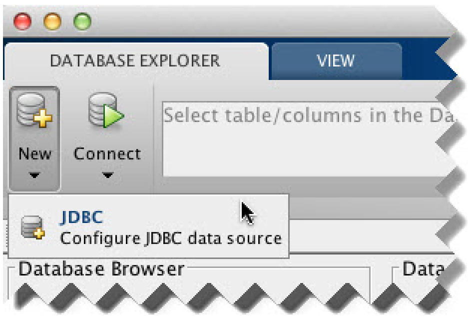
Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".

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

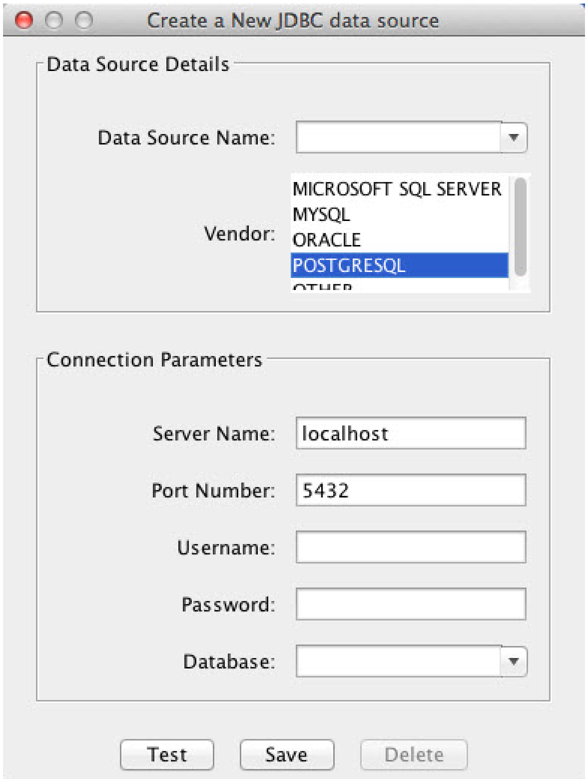
This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to PostgreSQL using the JDBC connection command line.” on page 2-151

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.

2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `POSTGRESQL` from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field.

5 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

6 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

7 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

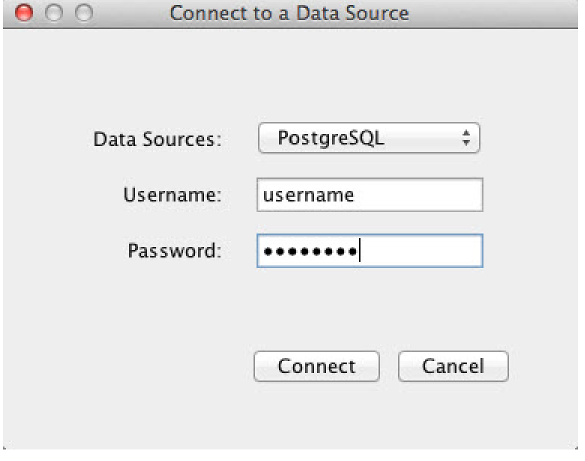
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

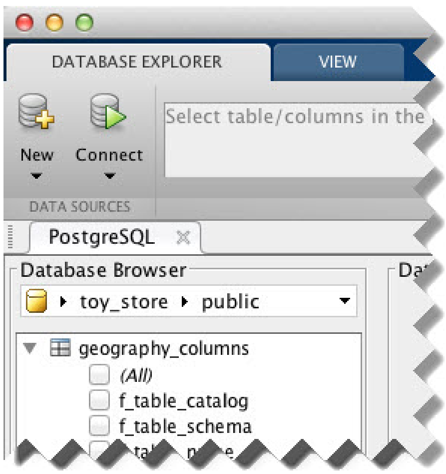
Step 4. Connect using Database Explorer or the command line.

Connect to PostgreSQL using Database Explorer.

1 After setting up the data source, connect to your database by selecting the data source name for the PostgreSQL database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

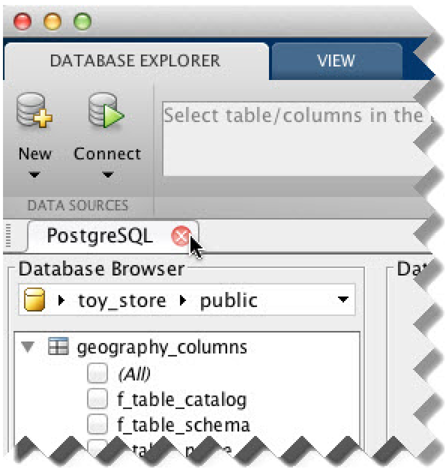


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button () next to the **PostgreSQL** data source name on the database tab. The **Close** button turns into a red circle (). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button () in the top-left corner.

If Database Explorer is docked, click the **Close** button () to close all database connections and Database Explorer.



Connect to PostgreSQL using the JDBC connection command line.

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

1. Use the `Vendor` name-value pair argument of `database` to specify a connection to a PostgreSQL database. For example, the following code assumes 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','PostgreSQL',...
                   'Server','sname');
   ```

2. Close the database connection `conn`.

   ```
   close(conn)
   ```

See Also

`close` | `database` | `javaaddpath`

More About

- “Working with Database Explorer” on page 4-2

- “Bring Java Classes into MATLAB Workspace”

PostgreSQL JDBC for Linux

This tutorial shows how to set up a data source and connect to your PostgreSQL database. This tutorial uses the JDBC4 PostgreSQL Driver, Version 8.4 to connect to the PostgreSQL 9.2 database.

In this section...

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` command in the Command Window. The output is a file path to a folder on your computer.
2 Close MATLAB if it is running.
3 Navigate to the folder and create a file called `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 and the JAR file name. For example, `/home/user/DB_Drivers/postgresql-8.4-702.jdbc4.jar`. Save and close `javaclasspath.txt`.
5 Restart MATLAB.

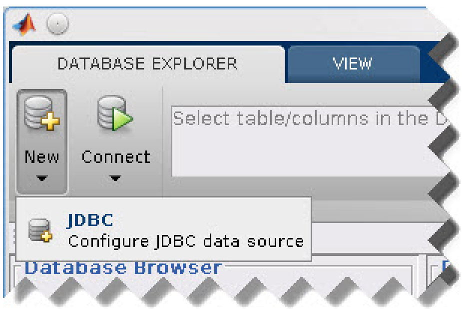
Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".

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

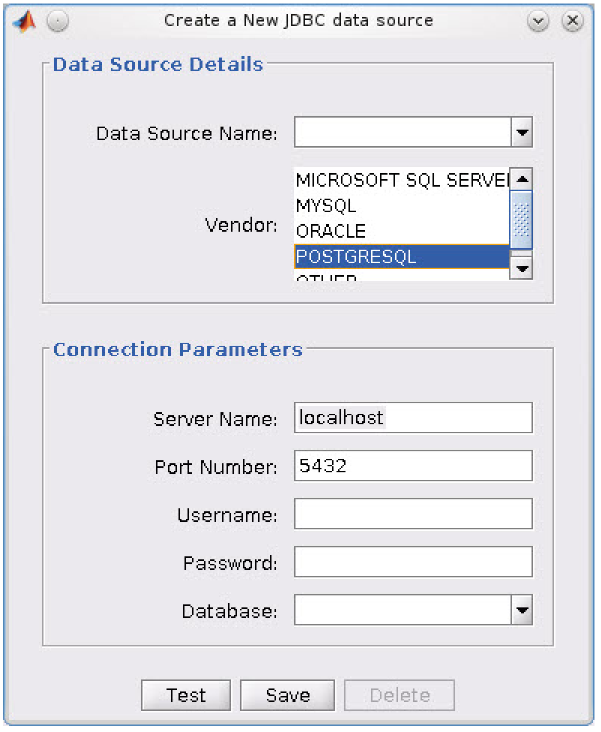
This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see "Connect to PostgreSQL using the JDBC connection command line." on page 2-158

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.

2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select POSTGRESQL from the **Vendor** list. After selecting the vendor, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

4 Enter the database server name in the **Server Name** field, port number in the **Port Number** field, user name in the **Username** field, password in the **Password** field, and database name in the **Database** field.

5 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

6 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

7 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

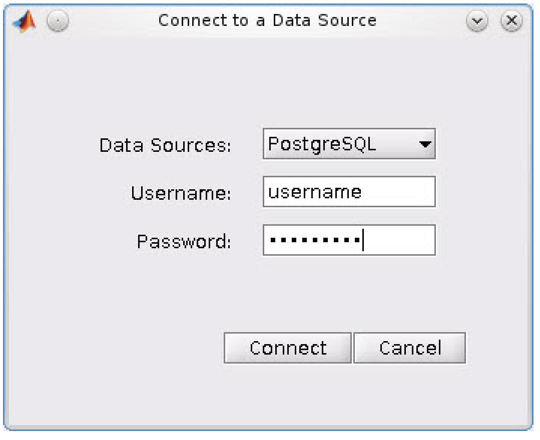
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

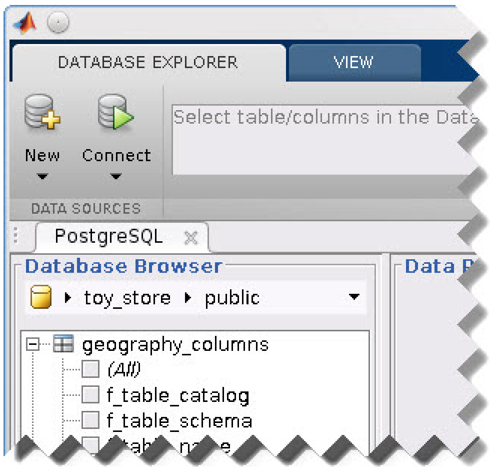
Step 4. Connect using Database Explorer or the command line.

Connect to PostgreSQL using Database Explorer.

1 After setting up the data source, connect to your database by selecting the data source name for the PostgreSQL database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

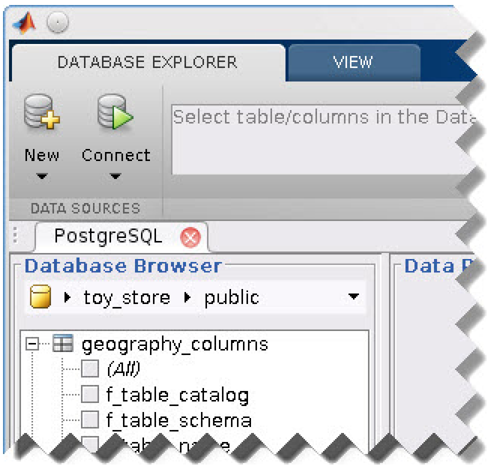


   Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **PostgreSQL** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to PostgreSQL using the JDBC connection command line.

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

1. Use the `Vendor` name-value pair argument of `database` to specify a connection to a PostgreSQL database. For example, the following code assumes 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','PostgreSQL',...
                'Server','sname');
```

2. Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2

- “Bring Java Classes into MATLAB Workspace”

SQLite JDBC for Mac OS X

This tutorial shows how to set up a data source and connect to your SQLite database. This tutorial uses the SQLite JDBC 3.7.2 Driver to connect to the SQLite Version 3.7.17 database.

Step 1. Verify the driver installation.

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

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 the MATLAB Interface to SQLite" on page 2-6.

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

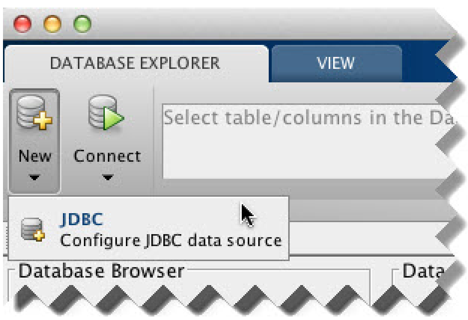
1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4. Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider and the JAR file name. For example, `/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 your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Bring Java Classes into MATLAB Workspace”.

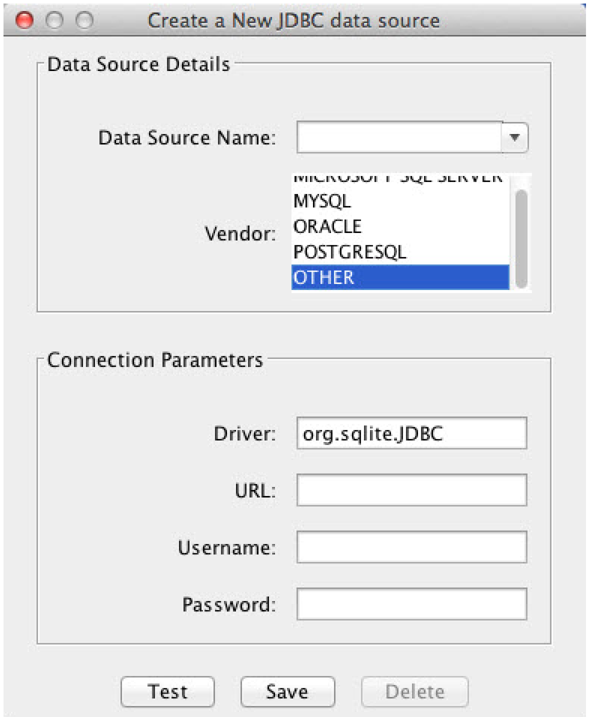
Step 3. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to SQLite using the JDBC connection command line.” on page 2-165

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New > JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `OTHER` from the **Vendor** list.

4 Enter the SQLite driver Java class object in the **Driver** field. Here, use `org.sqlite.JDBC`. After entering the driver, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

5 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, `subprotocol` is `sqlite`. The last part of the URL string is `subname`. For SQLite, this 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 into the **URL** field.

6 Enter your user name in the **Username** field and your password in the **Password** field, or leave them blank if your database does not need them.

7 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

8 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

9 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

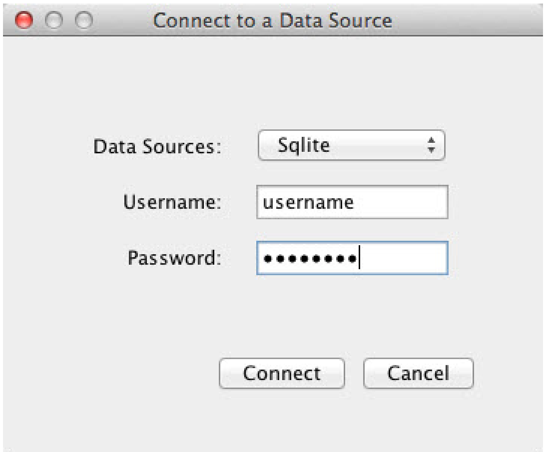
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

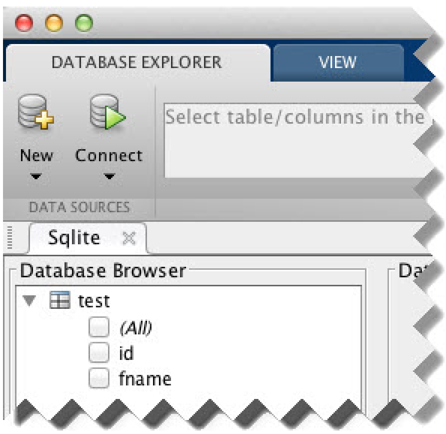
Step 4. Connect using Database Explorer or the command line.

Connect to SQLite using Database Explorer.

1 After setting up the data source, connect to your database by selecting the data source name for the SQLite database from the **Data Sources** list. Enter a user name and password or leave them blank if your database does not require them. Click **Connect**.

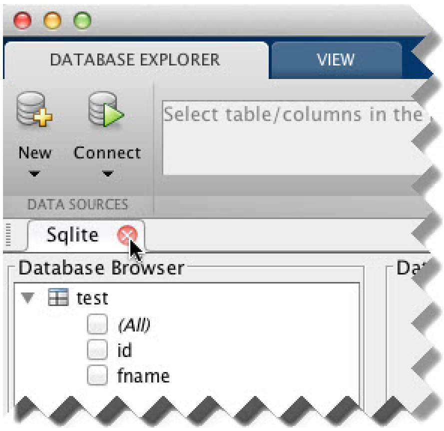


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button () next to the **Sqlite** data source name on the database tab. The **Close** button turns into a red circle (). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button () in the top-left corner.

If Database Explorer is docked, click the **Close** button (☒) to close all database connections and Database Explorer.



Connect to SQLite using the JDBC connection command line.

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

1. Create 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, `subprotocol` is `sqlite`. The last part of the URL string is `subname`. For SQLite, this 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.
2. Connect to the SQLite database by using the `database` function. Enter the full path to your SQLite database `dbpath` for the first argument, or leave this argument blank and include the full path in the URL string `URL`. Enter your user name `username` and your password `pwd`, or leave these blank if your database does not require them. The fourth argument is the driver Java class object. This code assumes the class object is `org.sqlite.JDBC`. The last argument is the URL string `URL`.

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

3. Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2
- "Bring Java Classes into MATLAB Workspace"

SQLite JDBC for Linux

This tutorial shows how to set up a data source and connect to your SQLite database. This tutorial uses the SQLite JDBC 3.7.2 Driver to connect to the SQLite Version 3.7.17 database.

Step 1. Verify the driver installation.

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

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 the MATLAB Interface to SQLite" on page 2-6.

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

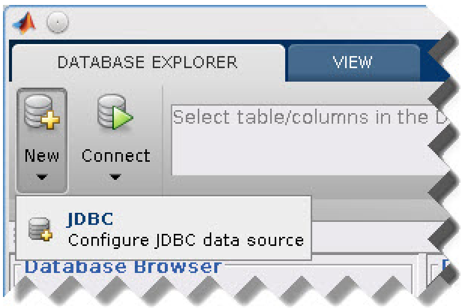
1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4. Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider and the JAR file name. For example, `/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 your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see “Bring Java Classes into MATLAB Workspace”.

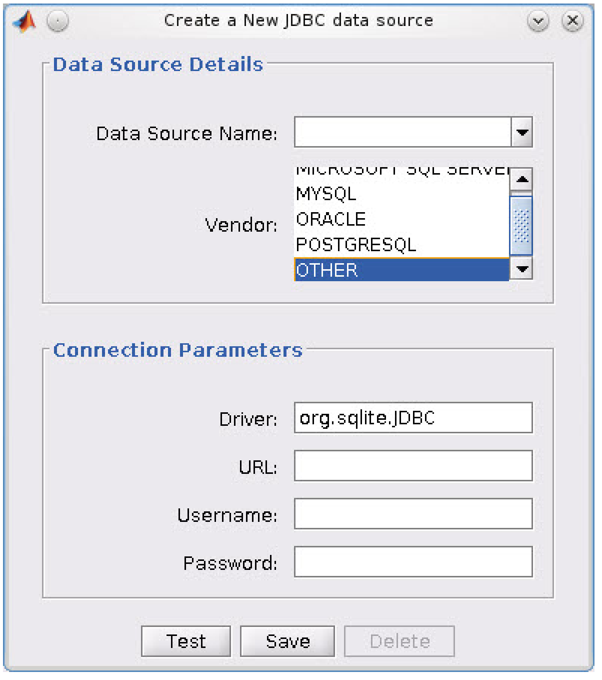
Step 3. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to SQLite using the JDBC connection command line.” on page 2-172

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `OTHER` from the **Vendor** list.

4 Enter the SQLite driver Java class object in the **Driver** field. Here, use `org.sqlite.JDBC`. After entering the driver, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

5 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, `subprotocol` is `sqlite`. The last part of the URL string is `subname`. For SQLite, this 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 into the **URL** field.

6. Enter your user name in the **Username** field and your password in the **Password** field, or leave them blank if your database does not need them.
7. Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!
8. Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.
9. If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

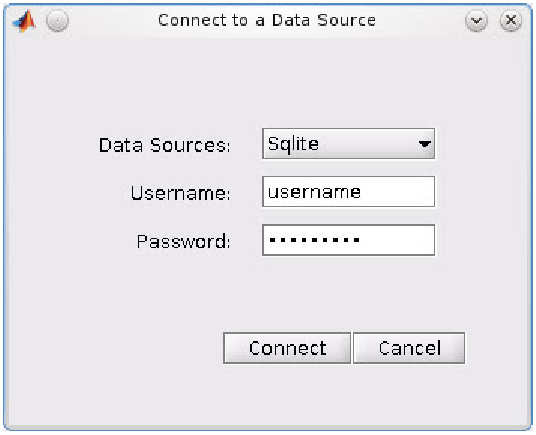
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

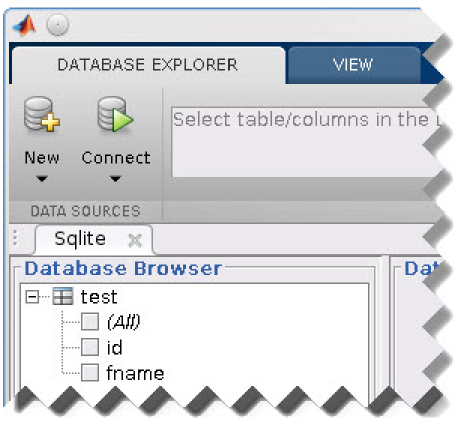
Step 4. Connect using Database Explorer or the command line.

Connect to SQLite using Database Explorer.

1. After setting up the data source, connect to your database by selecting the data source name for the SQLite database from the **Data Sources** list. Enter a user name and password or leave them blank if your database does not require them. Click **Connect**.

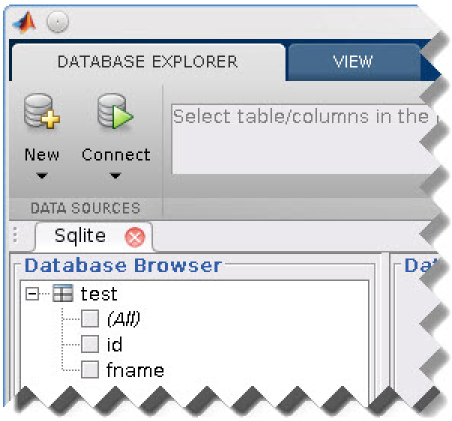


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **Sqlite** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button (X) to close all database connections and Database Explorer.



Connect to SQLite using the JDBC connection command line.

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

1 Create 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, `subprotocol` is `sqlite`. The last part of the URL string is `subname`. For SQLite, this 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.

2 Connect to the SQLite database by using the `database` function. Enter the full path to your SQLite database `dbpath` for the first argument, or leave this argument blank and include the full path in the URL string `URL`. Enter your user name `username` and your password `pwd`, or leave these blank if your database does not require them. The fourth argument is the driver Java class object. This code assumes the class object is `org.sqlite.JDBC`. The last argument is the URL string `URL`.

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

3 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2
- "Bring Java Classes into MATLAB Workspace"

Sybase JDBC for Mac OS X

This tutorial shows how to set up a data source and connect to your Sybase database. This tutorial uses the jConnect 4 JDBC Driver to connect to the Sybase Adaptive Server Enterprise 15.7 database.

In this section...

Step 1. Verify the driver installation.

If the JDBC driver for Sybase 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.

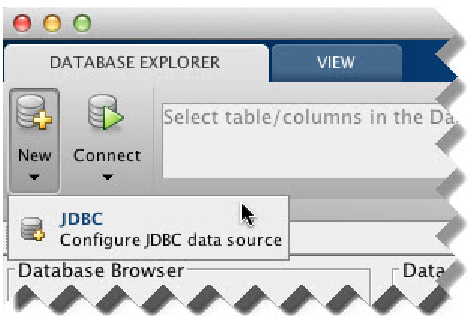
1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4. Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider and the JAR file name. For example, `/home/user/DB_Drivers/jconn4.jar`. Save and close `javaclasspath.txt`.
5. Restart MATLAB.

Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".

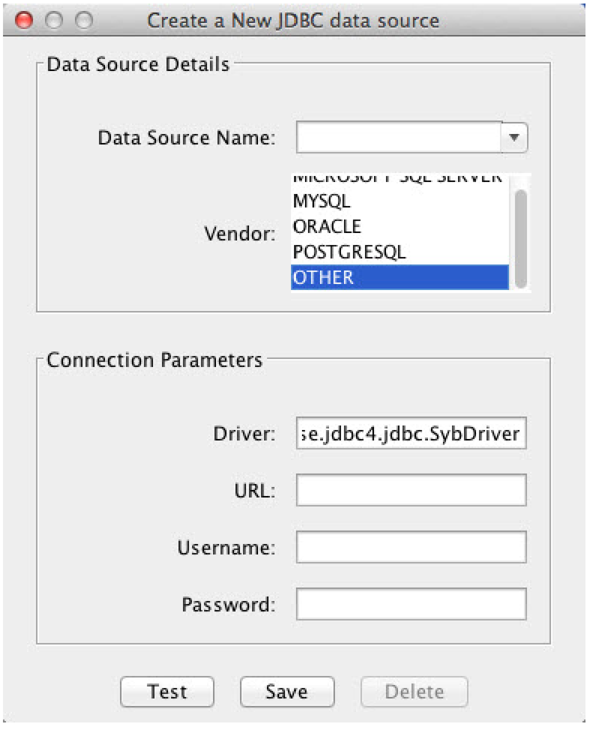
Step 3. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see “Connect to Sybase using the JDBC connection command line.” on page 2-179

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3. Select `OTHER` from the **Vendor** list.
4. Enter the Sybase driver Java class object in the **Driver** field. Here, use `com.sybase.jdbc4.jdbc.SybDriver`. After entering the driver, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

5. Connect to the Sybase 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, `subprotocol` is `sybase:Tds`. The last part of the URL string is `subname`. For Sybase, this contains the server name, the port number, and the database name. For example, your string

is `jdbc:sybase:Tds:ServerName:PortNumber/dbname`, where `ServerName` is your server name, `PortNumber` is your port number, and `dbname` is your database name. Enter your full string into the **URL** field.

6. Enter your user name in the **Username** field and your password in the **Password** field.
7. Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!
8. Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.
9. If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

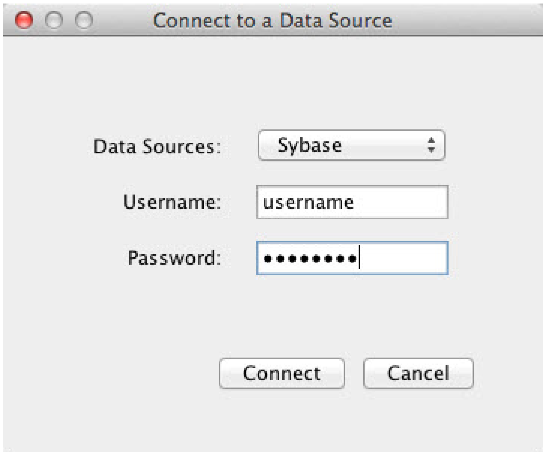
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

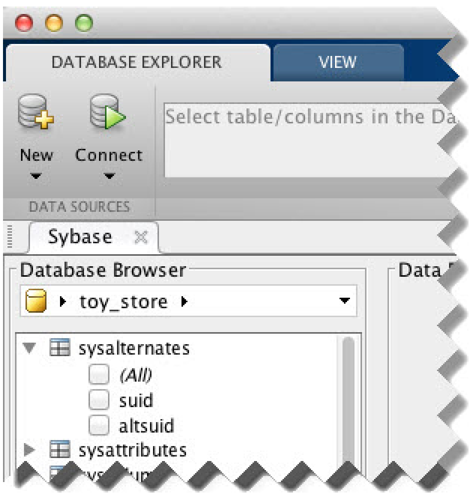
Step 4. Connect using Database Explorer or the command line.

Connect to Sybase using Database Explorer.

1. After setting up the data source, connect to your database by selecting the data source name for the Sybase database from the **Data Sources** list. Enter a user name and password. Click **Connect**.



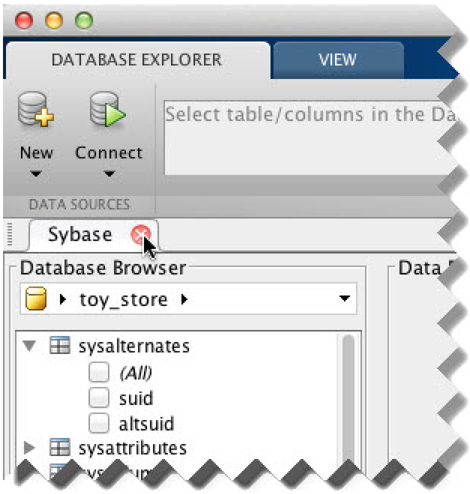
Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **Sybase** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you

want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-left corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Sybase using the JDBC connection command line.

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

1. Create 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, `subprotocol` is `sybase:Tds`. The last part of the URL string is `subname`. For Sybase, this contains the server name, the port number, and the database name. For example, your URL string is `jdbc:sybase:Tds:ServerName:PortNumber/dbname`, where `ServerName` is your server name, `PortNumber` is your port number, and `dbname` is your database name.
2. Connect to the Sybase database using the `database` function. For example, the following code assumes you are connecting to a database named `dbname` with user name `username` and password `pwd`. The fourth argument is the driver Java class

object. This code assumes the class object is `com.sybase.jdbc4.jdbc.SybDriver`. The last argument is the URL string `URL`.

```
conn = database('dbname','username','pwd',...
                'com.sybase.jdbc4.jdbc.SybDriver','URL');
```

3 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- “Working with Database Explorer” on page 4-2
- “Bring Java Classes into MATLAB Workspace”

Sybase JDBC for Linux

This tutorial shows how to set up a data source and connect to your Sybase database. This tutorial uses the jConnect 4 JDBC Driver to connect to the Sybase Adaptive Server Enterprise 15.7 database.

Step 1. Verify the driver installation.

If the JDBC driver for Sybase 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.

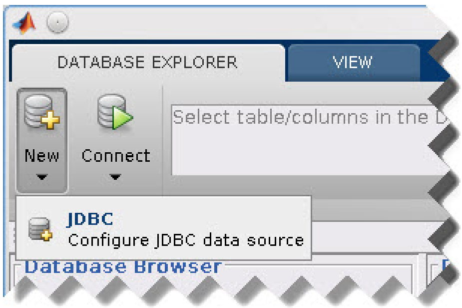
1. Run the `prefdir` command in the Command Window. The output is a file path to a folder on your computer.
2. Close MATLAB if it is running.
3. Navigate to the folder and create a file called `javaclasspath.txt` in the folder.
4. Open `javaclasspath.txt`. Add the full path to the database driver JAR file in `javaclasspath.txt`. The full path includes the path to the folder where you downloaded the JAR file from the database provider and the JAR file name. For example, `/home/user/DB_Drivers/jconn4.jar`. Save and close `javaclasspath.txt`.
5. Restart MATLAB.

Alternatively, you can use `javaaddpath` to add your JDBC driver to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".

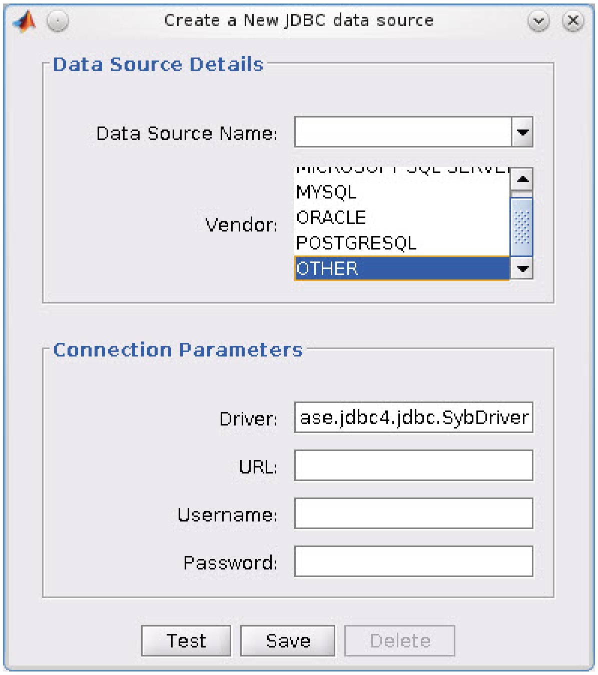
Step 3. Set up the data source using Database Explorer.

This step is required only for connecting to Database Explorer. If you want to use the command line to connect to your database, see "Connect to Sybase using the JDBC connection command line." on page 2-186

1 Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Then, select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, enter `dexplore` at the command line. If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
2 Click the **Database Explorer** tab, and then select **New** > **JDBC**.



The Create a New JDBC data source dialog box opens.



3 Select `OTHER` from the **Vendor** list.

4 Enter the Sybase driver Java class object in the **Driver** field. Here, use `com.sybase.jdbc4.jdbc.SybDriver`. After entering the driver, if you did not add the JDBC driver file path to the Java class path, this dialog box displays this message at the bottom. Address this message by following the steps described in Step 2.

5 Connect to the Sybase 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, `subprotocol` is `sybase:Tds`. The last part of the URL string is `subname`. For Sybase, this contains the server name, the port number, and the database name. For example, your string is `jdbc:sybase:Tds:ServerName:PortNumber/dbname`, where `ServerName` is your server name, `PortNumber` is your port number, and `dbname` is your database name. Enter your full string into the **URL** field.

6 Enter your user name in the **Username** field and your password in the **Password** field.

7 Click **Test** to test the connection. If your connection succeeded, Database Explorer displays Connection Successful!

8 Enter a data source name in the **Data Source Name** field in the Create a New JDBC data source dialog box. Use a new data source name that does not appear in the existing list of data source names. Click **Save**. The new JDBC data source appears in the list of data sources in the Connect to a Data Source dialog box.

9 If this time is the first time that you are creating a data source using Database Explorer, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

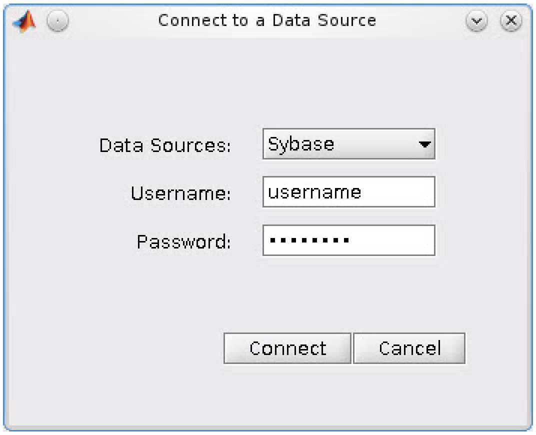
   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

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

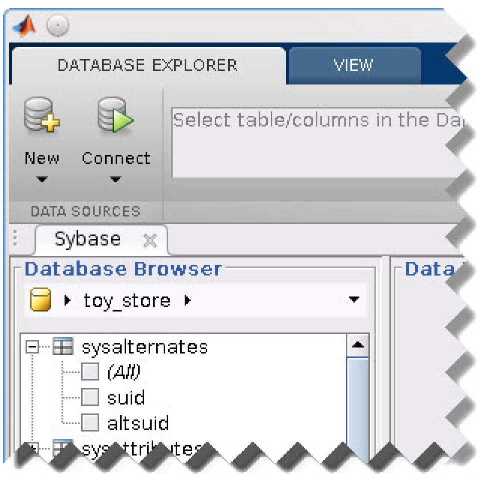
Step 4. Connect using Database Explorer or the command line.

Connect to Sybase using Database Explorer.

1 After setting up the data source, connect to your database by selecting the data source name for the Sybase database from the **Data Sources** list. Enter a user name and password. Click **Connect**.

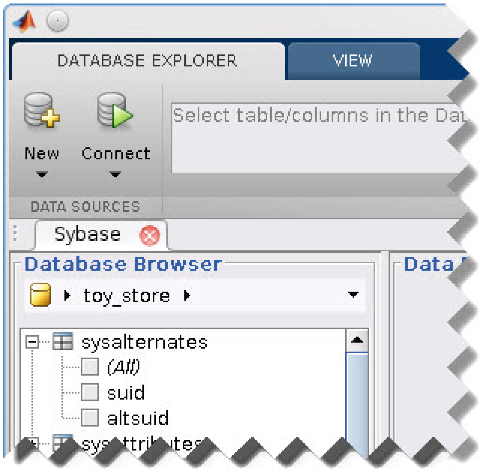


Database Explorer connects to your database and displays its contents in a tab named with the data source name.



2 Close the connection using Database Explorer by hovering the cursor over the **Close** button ( ) next to the **Sybase** data source name on the database tab. The **Close** button turns into a red circle ( ). Click it to close the database connection. If you want to close Database Explorer and all database connections, click the **Close** button ( ) in the top-right corner.

If Database Explorer is docked, click the **Close** button ( ) to close all database connections and Database Explorer.



Connect to Sybase using the JDBC connection command line.

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

1 Create 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, `subprotocol` is `sybase:Tds`. The last part of the URL string is `subname`. For Sybase, this contains the server name, the port number, and the database name. For example, your URL string is `jdbc:sybase:Tds:ServerName:PortNumber/dbname`, where `ServerName` is your server name, `PortNumber` is your port number, and `dbname` is your database name.

2 Connect to the Sybase database using the `database` function. For example, the following code assumes you are connecting to a database named `dbname` with user name `username` and password `pwd`. The fourth argument is the driver Java class object. This code assumes the class object is `com.sybase.jdbc4.jdbc.SybDriver`. The last argument is the URL string `URL`.

```
conn = database('dbname','username','pwd',...
```

```
                'com.sybase.jdbc4.jdbc.SybDriver','URL');
```

3 Close the database connection `conn`.

```
close(conn)
```

See Also

`close` | `database` | `javaaddpath`

More About

- "Working with Database Explorer" on page 4-2
- "Bring Java Classes into MATLAB Workspace"

Other ODBC- or JDBC-Compliant Databases

This tutorial provides high-level workflows for using other ODBC- or JDBC-compliant databases.

In this section...

ODBC-Compliant Databases

This tutorial shows how to configure your driver and connect to your ODBC-compliant database with MATLAB. Database Toolbox can connect to any ODBC-compliant database that is relational and that uses ANSI SQL. For example, if your database is Microsoft Excel or IBM DB2, here are some basic steps to follow.

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 Data Source ODBC Administrator.
2. Create a data source that uses the installed driver using the Microsoft Data Source ODBC Administrator. For details about the Microsoft Data Source ODBC Administrator, see Driver Installation.
3. Use Database Explorer to test your connection. For details, see "Configure ODBC Data Sources" on page 4-8.
4. Use Database Explorer to connect to your database. For details, see "Connect to a Data Source" on page 4-16.
5. Alternatively, you can connect to your database using the command line function `database`.
6. For more in-depth assistance, contact your database administrator or database documentation. For more in-depth instructions, see the example "MySQL ODBC for Windows" on page 2-59.

JDBC-Compliant Databases

This tutorial shows how to configure your driver and connect to your JDBC-compliant database with MATLAB. Database Toolbox can connect to any JDBC-compliant database that is relational and that uses ANSI SQL. For example, if your database is Apache™

Derby or Microsoft Windows Azure, here are some basic steps to follow. The details of the steps below can vary depending on your database and database version. For details about your database, contact your database administrator or refer to your database documentation. The driver and URL fields (in Database Explorer Create a New JDBC data source dialog box and in the database command) 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.

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 alternatively to the dynamic Java class path. For details about static and dynamic class paths, see "Bring Java Classes into MATLAB Workspace".
3. To connect to a JDBC-compliant database, you need to know your database driver Java class object. For example, the Java class object for a SQLite database is `org.sqlite.JDBC`. Use this value for establishing a connection either with Database Explorer in the **Driver** field or the command line in the `driver` argument.
4. To connect to a JDBC-compliant database, you need to 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. The `subprotocol` is the database type. The last part of the URL string is the `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, your 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 Database Explorer or the command line.
5. Use Database Explorer to test your connection. For details, see "Configure JDBC Data Sources" on page 4-12.
6. Use Database Explorer to connect to your database. For details, see "Connect to a Data Source" on page 4-16.
7. Alternatively, you can connect to your database using the command line function `database`.
8. For more in-depth assistance, contact your database administrator or database documentation. For more in-depth instructions, see the example "Sybase JDBC for Windows" on page 2-97.

See Also

`close` | `database`

Related Examples

- “MySQL ODBC for Windows” on page 2-59
- “Sybase JDBC for Windows” on page 2-97

More About

- “Bring Java Classes into MATLAB Workspace”
- “Working with Database Explorer” on page 4-2

Connecting to a Database

To connect to your database, install an ODBC or JDBC driver and define a data source. For details about driver installation and data source setup, see “Configuring a Driver and Data Source” on page 2-16. 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 the MATLAB Interface to SQLite” on page 2-6.

Connection Options

You can connect to your database using Database Explorer or the command line. You can perform different actions using these two options. For details about deciding which option to use, see “Connection Options” on page 2-10.

Microsoft Access

- ODBC
  - “Connect to Microsoft Access using Database Explorer.” on page 2-22
  - “Connect to Microsoft Access using the native ODBC connection command line.” on page 2-24

Microsoft SQL Server

- ODBC

  - “Connect to Microsoft SQL Server using Database Explorer.” on page 2-31
  - “Connect to Microsoft SQL Server using the native ODBC connection command line.” on page 2-33
- JDBC
  - “Connect to Microsoft SQL Server using Database Explorer.” on page 2-42
  - “Connect to Microsoft SQL Server using the JDBC connection command line.” on page 2-44

Oracle

- ODBC
  - Database Explorer cannot work with the Oracle ODBC driver because of an issue with the JDBC/ODBC bridge. For details, see “Database Explorer Error Messages” on page 3-15.
  - To connect using the command line, see “Step 3. Connect using the native ODBC connection command line.” on page 2-49
- JDBC
  - “Connect to Oracle using Database Explorer.” on page 2-54
  - “Connect to Oracle using the JDBC connection command line.” on page 2-56

MySQL

- ODBC
  - “Connect to MySQL using Database Explorer.” on page 2-62
  - “Connect to MySQL using the native ODBC connection command line.” on page 2-64
- JDBC
  - “Connect to MySQL using Database Explorer.” on page 2-68
  - “Connect to MySQL using the JDBC connection command line.” on page 2-70

PostgreSQL

- ODBC
  - “Connect to PostgreSQL using Database Explorer.” on page 2-74
  - “Connect to PostgreSQL using the native ODBC connection command line.” on page 2-76
- JDBC
  - “Connect to PostgreSQL using Database Explorer.” on page 2-80
  - “Connect to PostgreSQL using the JDBC connection command line.” on page 2-82

SQLite

- JDBC
  - “Connect to SQLite using Database Explorer.” on page 2-86
  - “Connect to SQLite using the JDBC connection command line.” on page 2-88
- No driver or database installation required. To create a SQLite connection, use `sqlite` to create a SQLite database file or connect to an existing SQLite database file.

Sybase

- ODBC
  - “Connect to Sybase using Database Explorer.” on page 2-94
  - “Connect to Sybase using the native ODBC connection command line.” on page 2-96
- JDBC
  - “Connect to Sybase using Database Explorer.” on page 2-100
  - “Connect to Sybase using the JDBC connection command line.” on page 2-102

Other ODBC- or JDBC-Compliant Databases

For an example of how to connect to a database that is not listed previously, see “Other ODBC- or JDBC-Compliant Databases” on page 2-188.

See Also

`close` | `database`

More About

- "Initial Setup Requirements" on page 2-12
- "Working with a Database and MATLAB" on page 2-3
- "Connection Options" on page 2-9
- "Choosing Between ODBC and JDBC Drivers" on page 2-13
- "Configuring a Driver and Data Source" on page 2-16
- "Working with the MATLAB Interface to SQLite" on page 2-6

Selecting Data

In this section...

You can open two different connections to the same database, one using Database Explorer and another using the command line. If you are working with large data sets, use the command line instead of Database Explorer for maximum performance. 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 the MATLAB Interface to SQLite" on page 2-6.

Use Database Explorer to Select Data

If you have minimal proficiency writing SQL queries or want to browse the data in your database quickly, use Database Explorer. To build queries using Database Explorer, see "Refine Results Using Query Criteria and Rules" on page 4-21.

Use the Command Line to Select Data

Exploring your database data using the command line requires knowledge of writing SQL queries. Use the `exec` and `fetch` functions to select data from your database. The `exec` function executes your SQL statement and the `fetch` function retrieves the data from the database into a MATLAB variable. If you are not comfortable with writing SQL, then use Database Explorer to select data from your database.

If you have a stored procedure you want to run using Database Toolbox, you can use the `runstoredprocedure` or `exec` function.

Working with Custom Data Types

Database Toolbox functions return custom data types, for example 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 your Java object. Use the available methods to retrieve data from your Java object. The steps for your object are specific to your database. For details, refer to your JDBC driver or database-specific documentation.

Running SQL Queries Saved in Scripts or Files

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

More About

- "Connection Options" on page 2-9
- "Connecting to a Database Using the Native ODBC Interface" on page 3-18
- "Working with Large Data Sets" on page 2-199
- "Working with the MATLAB Interface to SQLite" on page 2-6

Inserting Data Using the Command Line

You can use `datainsert`, `fastinsert`, or `insert` to insert data using the command line. Only `insert` is supported for the MATLAB interface to SQLite. To understand which function is best for your purposes and setup, see this table.

	`datainsert`	**`fastinsert`**	**`insert`**
Native ODBC connection	All three functions have identical functionality.		
JDBC or JDBC/ ODBC bridge connection	Faster performance.	Both `fastinsert` and `insert` have identical functionality.	
Special formatting	`datainsert` requires special formatting for dates and timestamps, `null`, and `NaN`.	Both `fastinsert` and `insert` require special formatting for dates and timestamps.	
Database preferences	`NullNumberWrite` and `NullStringWrite` database preference settings do not apply.	All database preference settings apply for both `fastinsert` and `insert`.	
MATLAB interface to SQLite	Not supported.	Not supported.	Supported.

If you still experience performance issues using these functions, then use the bulk insert functionality of your database. For details, see "Export Data Using Bulk Insert" on page 6-31.

To fetch data in your database, use the `exec` and `fetch` functions.

To set database preferences, use the `setdbprefs` function.

See Also
`database`

Related Examples

- "Export Data to New Record in Database" on page 6-22

- “Export Multiple Records from the MATLAB Workspace” on page 6-27
- “Export Data Using Bulk Insert” on page 6-31
- “Import Data Using the MATLAB® Interface to SQLite” on page 6-75

More About

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

Working with Large Data Sets

In this section...

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 a Database Using the Native ODBC Interface" on page 3-18. If the native ODBC interface does not work, connect to your database using a JDBC driver. For details, see "Connecting to a Database" on page 2-191.

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`.

When working with a JDBC connection, you can run into out-of-memory issues because of JVM heap memory restrictions. To achieve the best performance with importing large sets of data into MATLAB, you can fetch the data in batches by setting database preferences. To assess your memory needs and for options on running an SQL query that returns large amounts of data, see "Preference Settings for Large Data Import" on page 5-19.

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 the 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 the 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”.

Related Examples

- “Import Data from Databases into MATLAB” on page 6-4
- “Export Data to New Record in Database” on page 6-22

More About

- “Choosing Between ODBC and JDBC Drivers” on page 2-13
- “Selecting Data” on page 2-195

Deploying a Database Application with MATLAB Compiler

In this section...

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 a 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".
2. Compile your database application with MATLAB Compiler using the standalone application packaging process. For details, see "Package Standalone Application with Application Compiler App".
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 2-35.

Caution: Do not add driver JAR files using `javaclasspath` as this might cause issues on the target machine. For details, see “Bring Java Classes into MATLAB Workspace”.

See Also

`javaaddpath`

More About

- “Write Deployable MATLAB Code”
- “Create Functions in Files”
- “Package Standalone Application with Application Compiler App”
- “Bring Java Classes into MATLAB Workspace”

Working with Data Sources

Setting Up ODBC Data Sources

For instructions on setting up ODBC data sources, see “Configuring a Driver and Data Source” on page 2-16.

Setting Up JDBC Data Sources

For instructions on setting up JDBC data sources, see “Configuring a Driver and Data Source” on page 2-16.

Accessing Existing JDBC Data Sources

To access an existing data source from Visual Query Builder in future MATLAB sessions:

1. In Visual Query Builder, select **Query** > **Define JDBC data source**.
2. In the Define JDBC data sources dialog box, click **Use Existing File**.
3. In the Specify Existing JDBC data source MAT-file dialog box, select the MAT-file that contains the data sources you want to use and click **Open**.

   The data sources in the selected MAT-file appear in the Define JDBC data sources dialog box.
4. Click **OK** to close the Define JDBC data sources dialog box. The data sources now appear in the Visual Query Builder **Data source** list.

Modifying Existing JDBC Data Sources

1. Access the existing data source as described in "Accessing Existing JDBC Data Sources" on page 3-4.
2. Select the data source in the Define JDBC Data Sources dialog box.
3. Modify the data in the **Driver** and **URL** fields.
4. Click **Add/Update**.
5. Click **OK** to save your changes and close the Define JDBC data sources dialog box.

Removing JDBC Data Sources

1. Access the existing data source as described in "Accessing Existing JDBC Data Sources" on page 3-4.
2. Click **Remove**.
3. Click **OK** to save your changes and close the Define JDBC data sources dialog box.

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 Database Explorer or the command line when using `exec` or `fetch`.

Vendor	Error Message	Probable Causes	Resolution
Microsoft SQL Server	The statement did not return a result set.	There are other SQL statements in the middle of the stored procedure. This error happens after executing `exec` but before executing `fetch`. This error happens only with the command line.	Add `'SET NOCOUNT ON'` at the beginning of your stored procedure. For details, see `exec`.
Microsoft SQL Server	JDBC Driver 3.0 returns incorrect date values when used with JRE™ 1.7 by a Java application.	There is an issue with the Microsoft SQL Server JDBC Driver 3.0. This error happens after executing `fetch`. This error happens either with Database Explorer or the command line.	Install a hotfix from Microsoft for JDBC Driver 3.0. Alternatively, upgrade your Microsoft SQL Server JDBC driver to version 4.0.
Microsoft SQL Server	Connection is busy with results for another command.	You are connecting to Microsoft SQL Server using a driver that `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	Write custom MATLAB code to process the Java objects into MATLAB variables.

Vendor	Error Message	Probable Causes	Resolution
		error happens after executing `fetch`. This error happens only with the command line.	

PostgreSQL	Java exception occurred: java.lang.OutOfMemo
	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 `exec` but before executing `fetch`. This error happens either with Database Explorer or the command line.	Write custom code. Write the code for connecting to your database via the command line. Then write the following.
			
			`set(conn,'AutoCommit','off`
			
			`h = conn.Handle;`
			
			`stmt = h.createStatement();`
			
			`stmt.setFetchSize(50);`
			
			`rs = stmt.executeQuery(java.lan * from largeData where productnumber <= 3000000'));`
			
			Modify the previous statement to include your SQL query instead.
			
			Then process the resultset object `rs` in batches.

See Also

`exec` | `fetch`

More About

- “Working with Database Explorer” on page 4-2
- “Selecting Data” on page 2-195
- “Call a Stored Procedure That Returns Data” on page 6-44

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	• Database Toolbox software is not installed. • You are connecting using the native ODBC interface with MATLAB R2013a or earlier.	• 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.	• 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.	Verify that you add the path to your JDBC driver to the static or dynamic path. Ensure that you provide the correct JDBC driver name for the `driver` and `databaseurl` arguments.
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.	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 a Driver

Vendor	Error Message	Probable Causes	Resolution
			and Data Source” on page 2-16.
Microsoft Access	[Microsoft][ODBC Microsoft Access Driver] ‘(unknown)’ is not a valid path. make sure that the path name is spelled correctly and that you are connected to the server on which the file resides	Error occurs in the Connection Failure dialog box after clicking **Connect** in the Connect to a Data Source dialog box. The file location of the Microsoft Access database is incorrect.	Verify the location of the database file. If the database file is on a network drive, map to the network drive. Modify the existing file location by selecting **New > ODBC** and selecting the existing database name from the ODBC Data Source Administrator dialog box. Then select **Configure** to change the database file location.
Microsoft SQL Server	The TCP/IP connection to the host `hostname`, port `portnumber` 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 2-35
Microsoft SQL Server	This driver is not configured for integrated authentication.	The Microsoft SQL Server Windows authentication library is not added to `javalibrarypath.txt`	Add the Microsoft SQL Server Windows authentication library to `javalibrarypath.txt`. For details about configuring a

Vendor	Error Message	Probable Causes	Resolution
			Microsoft SQL Server Authenticated Database Connection, see "Microsoft SQL Server JDBC for Windows" on page 2-35.
Microsoft SQL Server or Sybase	Invalid string or buffer length.	64-bit ODBC driver error.	Use a JDBC driver or the native ODBC interface instead.
Microsoft SQL Server	JDBC Driver Error: com.microsoft.sqlserver.jdb Not Found/Loaded.	The full path to the JAR file was not added to the `javaclasspath.txt` file, or it was added using the `javaaddpath` 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.jdb <clinit> WARNING: Failed to load the sqljdbc_auth.dll	The path to the folder containing the file `sqljdbc_auth.dll` was not added to the `javalibrarypath.txt` 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 `sqljdbc_auth.dll` to the `javalibrarypath.txt` file. For details about configuring a Microsoft SQL Server Authenticated Database Connection, see "Microsoft SQL Server JDBC for Windows" on page 2-35.

Vendor	Error Message	Probable Causes	Resolution
Microsoft SQL Server	Login failed for user 'DOMAIN\username'.	Either the login credentials you are using are incorrect or your user account does not have enough rights to access the remote machine. This error also occurs when the database server is not configured to accept Integrated Windows Authentication login credentials.	Ensure that your user name and password are correct. Refer to your system administrator for appropriate access rights to your machines. Contact your database administrator to see if your database is set up with Windows Authentication.
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.

Oracle	Error when connecting to Oracle `oci8` database using JDBC driver:
	`Error using com.mathworks.toolbox`
	`Java exception occurred:`
	`java.lang.UnsatisfiedLinkError: n`
	`java.library.pathat`
	`java.lang.ClassLoader.loadLibrary`
	`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 `javalibrarypath.txt`. For details, see "Oracle JDBC for Windows" on page 2-50.

Oracle	Invalid Oracle URL specified:	The `DriverType` parameter is not specified.	Specify the `DriverType` parameter as either `thin` for connecting

Vendor	Error Message	Probable Causes	Resolution
	`OracleDataSource.makeURL`		without Windows authentication or `oci` for connecting with Windows authentication.
Oracle	The Network Adapter could not establish the connection.	Either `Server` or `Portnumber` 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.

See Also

`database`

More About

- “Configuring a Driver and Data Source” on page 2-16
- “Connecting to a Database Using the Native ODBC Interface” on page 3-18

Database Explorer Error Messages

This table describes how to address common errors you might encounter while working with Database Explorer. For details about Database Toolbox connection errors, see "Database Connection Error Messages" on page 3-10.

Database Explorer Error Messages and Probable Causes

Vendor	Error Location	Error Message	Probable Causes	Resolution
All JDBC-Complian Database	Error occurs in the Connectio Failure dialog box after clicking **Connect** in the Connect to a Data Source dialog box.	[Microsoft][ODBC Driver Manager] Data source name not found and no default driver specified	JDBC data sources created by Visual Query Builder cannot be used in Database Explorer.	You must run this command: `setdbprefs(..., 'JDBCDataSourceFile',... '')` Then, create a new JDBC data source from Database Explorer.
Microsoft SQL Server	Error occurs in the Data Preview Error dialog box after selecting a column of a table in the	Invalid Object Name catalog name.table name	The selected schema name in Database Explorer is incorrect.	You must select the appropriate schema name in Database Explorer using the **Catalog/Schema** address bar above the table columns tree.

Vendor	Error Location	Error Message	Probable Causes	Resolution
	Database Browser pane.			
Oracle	Error occurs inside the Database Browser pane.	No tables found in this schema Consider changing the schema.	Database Explorer has a conflict with the Oracle ODBC driver due to issues in the JDBC/ODBC bridge.	Switch your database connection to use a JDBC driver. For details, see "Configuring a Driver and Data Source" on page 2-16.
Oracle	Error occurs after clicking **Connect** in the Connect to a Data Source dialog box.	Unable to get meta data:[Oracle][ODBC]Driver not capable.	Database Explorer has a conflict with the Oracle ODBC driver due to issues in the JDBC/ODBC bridge.	Switch your database connection to use a JDBC driver. For details, see "Configuring a Driver and Data Source" on page 2-16.
Oracle	Error occurs after trying to change the schema using Oracle ODBC driver.	Error changing catalog/schema: [Oracle][ODBC]Driver not capable	Database Explorer has a conflict with the Oracle ODBC driver due to issues in the JDBC/ODBC bridge.	Switch your database connection to use a JDBC driver. For details, see "Configuring a Driver and Data Source" on page 2-16.

More About

- “Working with Database Explorer” on page 4-2

Connecting to a Database Using the Native ODBC Interface

In this section...

About the Native ODBC Interface

The native ODBC interface is a C++ library that allows direct communication with the ODBC driver instead of using the Oracle JDBC/ODBC bridge. This interface eliminates issues from using the bridge and eliminates heap memory outages caused by the JVM heap memory restrictions. Using the native ODBC interface results in an improved data import and export experience, especially when working with 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 the Database Using the Native ODBC Interface

Connect to the database with the ODBC data source name, `dbtoolboxdemo`, using the user name, `admin`, and password, `admin`.

```
conn = database.ODBCConnection('dbtoolboxdemo','admin','admin');
```

`database.ODBCConnection` returns `conn` as a `database.ODBCConnection` object.

Import Data Using the Native ODBC Interface

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

```
curs = exec(conn,'select productDescription from productTable');
```

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

Note: The native ODBC interface has a default batch size of 100,000 that enables acceptable performance. To override this value, use `setdbprefs` as follows. Set `FetchInBatches` to `yes` and set `FetchBatchSize` to a specific batch size number `<h>`.

```
setdbprefs('FetchInBatches','yes')
setdbprefs('FetchBatchSize','<h>')
```

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

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

View the contents of the `Data` element in the cursor object `curs`.

```
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 the 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 `productTable` with the defined column names, `colnames`.

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

Caution: The Microsoft Access ODBC driver demonstrates unexpected behavior during large inserts. When inserting large amounts 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 `conn`.

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

Database Connection Type Comparison

You can connect to your database using:

- Native ODBC interface
- JDBC driver
- JDBC/ODBC bridge

Note: The JDBC/ODBC bridge functionality will be removed in a future release. To connect to a database, use the native ODBC interface or a JDBC driver instead.

This table highlights the differences between using these connection types to access and manipulate data in a database.

Item	Native ODBC Interface	JDBC Driver	JDBC/ODBC Bridge
Connection function	Use `database.ODBCConn`	Use `database`	Use `database`
Actions	See the supported functions list for actions and corresponding functions	Can perform the following actions: • Query data (`exec`) • Import data (`fetch`) • Export data (`datainsert`, `fastinsert`, `insert`, `update`) • Run stored procedure (`exec`, `runstoredproce` • Retrieve metadata (`dmd`, `tables`, `columns`, `catalogs`, and many others) • Use Database Explorer • Close connection (`close`)	Can perform the same actions as JDBC
Underlying technology	C++	Java	Java
Memory performance	Restricted by MATLAB memory, but not JVM heap memory	Restricted by both JVM heap memory and MATLAB memory	Restricted by both JVM heap memory and MATLAB memory
Data access performance	Fastest	Medium	Slowest

Item	Native ODBC Interface	JDBC Driver	JDBC/ODBC Bridge
64-bit systems	No major issues	No major issues	Several known issues with connectivity and data access
Data type support	Long data types are not supported (e.g. `LONG`, `BLOB`, etc.)	Long data types are supported	Long data types are supported

For details about the `database.ODBCConnection` syntax, see the native ODBC interface example in `database`. For details about choosing which connection type is best for your situation, see "Choosing Between ODBC and JDBC Drivers" on page 2-13.

Compatibility and Limitations

The native ODBC interface has the following compatibility and limitation considerations:

- The native ODBC interface is available only for the command line. You cannot use Database Explorer to access the native ODBC interface.
- The native ODBC interface does not support these functions:
  - `dmd`
  - `resultset`
  - `rsmd`
  - `runstoredprocedure`
- The native ODBC interface does not support long data types such as Oracle `LONG` and SQL Server `NTEXT`. If one of these errors displays, you are accessing an unsupported data type:
  - Driver unable to retrieve length for column number: <index of column in the query>
  - Out of memory. Type HELP MEMORY for your options.

More About

- "Connection Options" on page 2-9
- "Selecting Data" on page 2-195

4

Using Database Explorer

Working with Database Explorer

If you are using Database Explorer for the first time, migrate from Visual Query Builder (VQB) and set Database Explorer preferences. After performing these tasks, you are ready to configure data sources and connect to your database.

Getting Started with Database Explorer

Database Explorer is a Database Toolbox app for connecting to a database and interacting with database data. You can configure ODBC and JDBC data sources, modify database connections, and work with multiple databases. You can also display data from a database table, explore data from multiple database tables, and import data to the MATLAB workspace for analysis. Once you create queries using Database Explorer, you can generate code. For details, see:

- "Configure Data Sources and Connect to Databases" on page 4-6
- "Refine Results Using Query Criteria and Rules" on page 4-21
- "Generate SQL and MATLAB Code" on page 4-25

Migrate from VQB to Database Explorer

Database Explorer replaces VQB as an app for exploring the data in your database. If you are using VQB, to help migrate from VQB to Database Explorer, refer to these points:

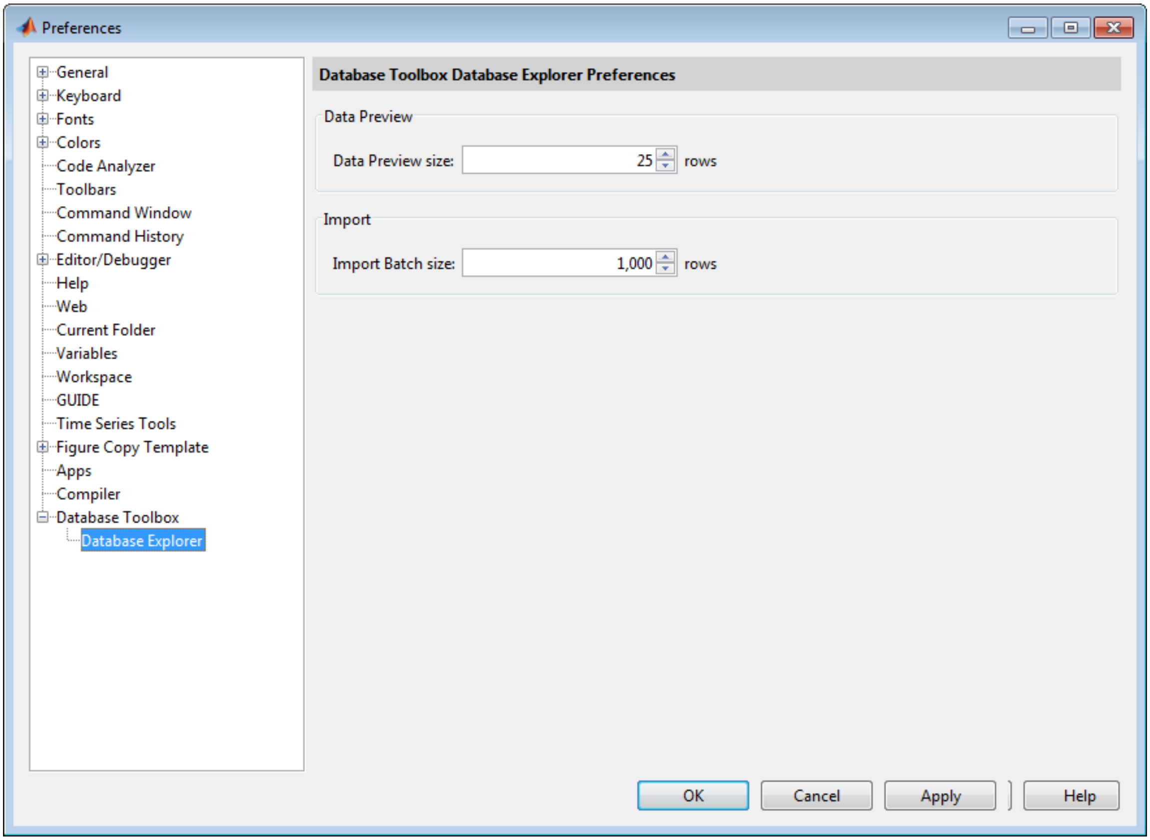
- If you use VQB (`querybuilder`) to access a JDBC data source, before starting Database Explorer for the first time, execute this command. You cannot use the same JDBC data source with Database Explorer.

  ```
  setdbprefs('JDBCDataSourceFile','')
  ```
  Then, define your JDBC data source using Database Explorer.

- If you use VQB to export data from MATLAB to your database, use the command-line functions `datainsert` or `fastinsert`.
- If you use VQB to generate reports, use MATLAB reporting and plotting functionality to generate reports. You can also use MATLAB Report Generator™ to generate reports.
- If you use VQB to display charts, use the MATLAB plotting tools to generate charts and graphics.
- If you generate MATLAB files using VQB, open Database Explorer and recreate your SQL query. Use Database Explorer to generate a script (`.m` file) that includes your SQL query, preference settings, and connection.
- If you save your SQL queries using VQB, open Database Explorer and recreate your SQL query. Use Database Explorer to generate a script with just your SQL query. Save the SQL script file with a `.sql` extension in MATLAB.

Set Database Explorer Preferences

1 On the Database Explorer Toolstrip, select **Preferences** to open the Database Explorer Preferences dialog box. These preference settings apply only to Database Explorer.



2 Specify the **Preferences** settings that apply to Database Explorer as described in the following table.

Preference	Allowable Values	Description
Data Preview size	5–10,000 rows	The number of rows that you see in the **Data Preview** pane of Database Explorer.
Import batch size	1,000–1,000,00 rows	The number of rows fetched at one time from a database. When importing large amounts of data using Database Explorer, tune this value for optimum performance. For details, see "Preference Settings for Large Data Import" on page 5-19.

Select **Database Toolbox** to manage additional preferences for Database Toolbox. For details, see "Working with Preferences" on page 5-15. Alternatively, use `setdbprefs` to specify preferences for the retrieved data.

3 Click **OK**.

See Also

`database` | `setdbprefs`

More About

- "Working with a Database and MATLAB" on page 2-3
- "Preference Settings for Large Data Import" on page 5-19
- "Working with Preferences" on page 5-15
- "Configure Data Sources and Connect to Databases" on page 4-6
- "Refine Results Using Query Criteria and Rules" on page 4-21
- "Generate SQL and MATLAB Code" on page 4-25

Configure Data Sources and Connect to Databases

In this section...

To make connections, Database Explorer uses data sources to identify your databases. Configure data sources to start exploring data in your databases. The data source setup differs depending on the database drivers that you are using for connection. Once data sources are available, you can connect to your databases, modify and delete database connections, and work with multiple databases at once.

Configure Your Environment

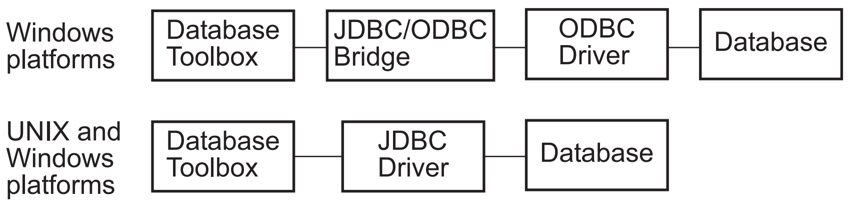
Before using Database Explorer to connect to a database, set up a *data source*. A data source consists of:

- Data that the toolbox accesses
- Information required to find the data, such as driver, folder, server, or network names

Data sources interact with *ODBC drivers* or *JDBC drivers*. An ODBC driver is a standard Microsoft Windows interface that enables communication between database management systems and SQL-based applications. A JDBC driver is a standard interface that enables communication between applications based on Oracle Java and database management systems.

Database Toolbox software is based on Java. It uses a JDBC/ODBC bridge to connect to the ODBC driver of a database, which the software installs as part of the MATLAB JVM.

This figure illustrates how drivers interact with Database Toolbox software.



Tip Some Windows systems support both ODBC and JDBC drivers. On such systems, JDBC drivers generally provide better performance than ODBC drivers because the software does not use the JDBC/ODBC bridge to access databases.

Before You Begin

Before you can use Database Explorer with the examples, do the following:

1. Set up the data sources that are provided with Database Toolbox.

   Requirement: If you use Visual Query Builder (`querybuilder`) to access a JDBC data source, before starting Database Explorer for the first time, execute this command. You cannot use the same JDBC data source with Database Explorer.

   ```
   setdbprefs('JDBCDataSourceFile','')
   ```

   ---

2. Configure the data sources for use with your database driver.

   - If you are using an ODBC driver, see "Configure ODBC Data Sources" on page 4-8.
   - If you are using a JDBC driver, see "Configure JDBC Data Sources" on page 4-12.

Set Up the `dbtoolboxdemo` Data Source

The `dbtoolboxdemo` data source uses the `tutorial` database located in `*matlabroot*/toolbox/database/dbdemos/tutorial.mdb`.

1. Copy `tutorial.mdb` into a folder to which you have write access.
2. Confirm that you have write access to `tutorial.mdb`.
3. Open `tutorial.mdb` from the MATLAB current folder by right-clicking the file and selecting **Open Outside MATLAB**. The file opens in Microsoft Access.

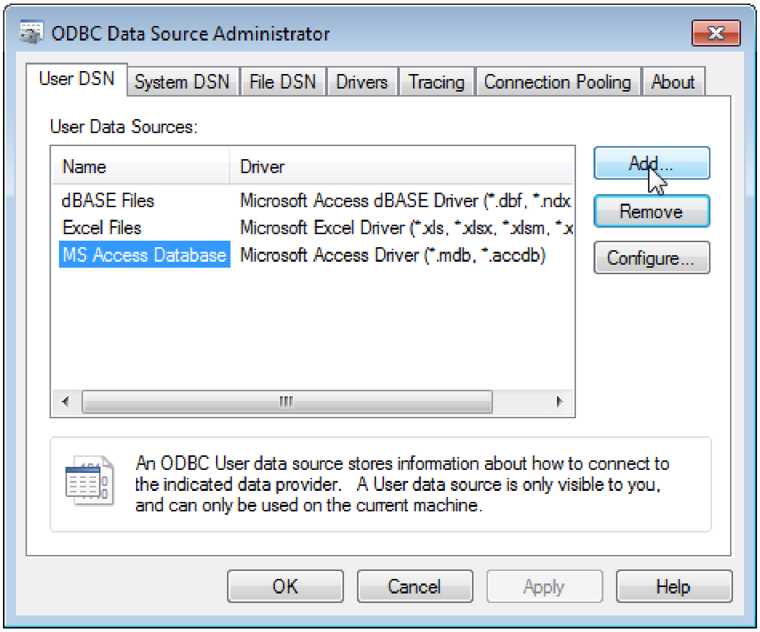
   Note: Depending on the Access version that you are running, you might need to convert the database to that version. For example, beginning in Microsoft Access 2007, you see the option to save as `*.accdb`. For details, consult your database administrator.

Configure ODBC Data Sources

When setting up a data source for an ODBC driver, the target database can be on a PC running the Windows operating system. Or, the target database can be on another system to which the PC is networked. These instructions use the Microsoft ODBC Data Source Administrator Version 6.1 for the U.S. English version of Microsoft Access 2010 for Windows systems. If you have a different configuration, consult your database administrator for help with your data source setup.

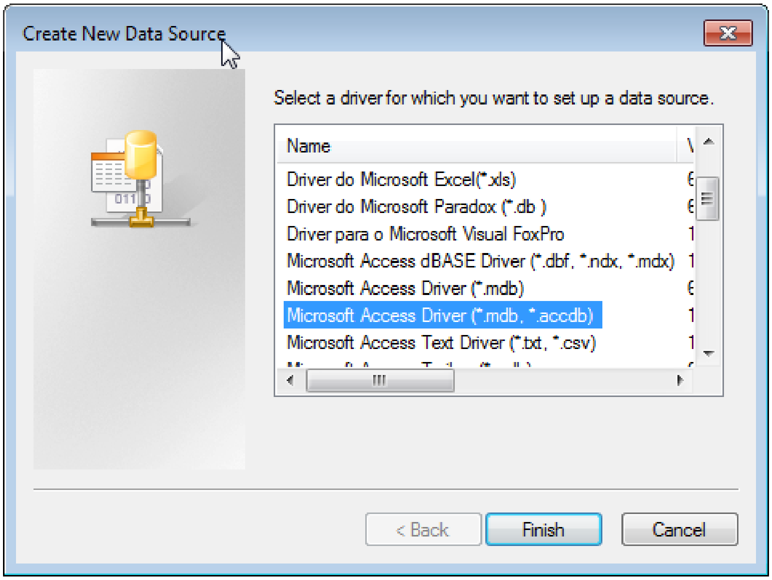
1. Close open databases, including `tutorial.mdb` in the database program.
2. Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, at the command line, enter:

   ```
   dexplore
   ```
   If no data sources are set up, a message box opens. Click **OK** to close it. Otherwise, the Connect to a Data Source dialog box opens. Click **Cancel** to close this dialog box.
3. Click the **Database Explorer** tab. Select **New** > **ODBC** to open the ODBC Data Source Administrator dialog box to define the ODBC data source.
4. Click the **User DSN** tab and click **Add**.

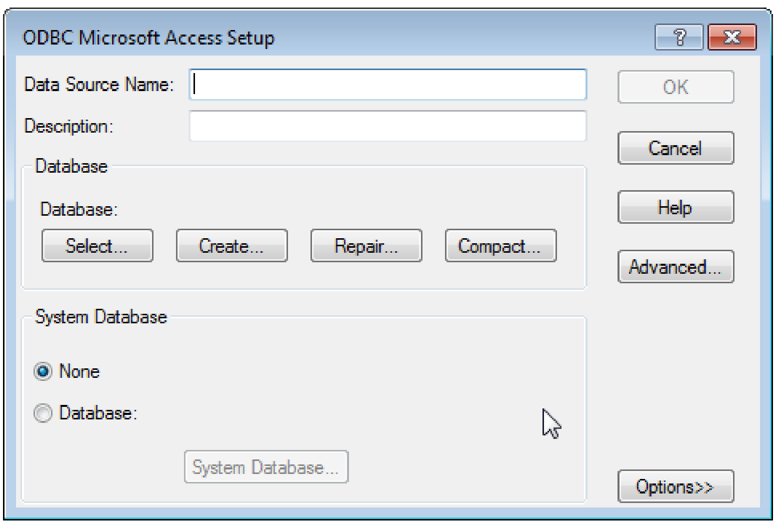


A list of installed ODBC drivers appears in the Create New Data Source dialog box.

5 Select `Microsoft Access Driver (*.mdb, *.accdb)` and click **Finish**.



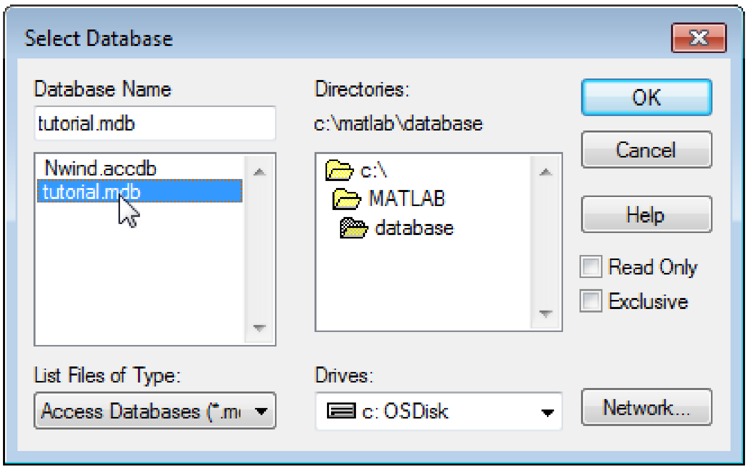
The ODBC Microsoft Access Setup dialog box for your driver opens. The dialog box for your driver can differ from the following dialog box.



6 Enter `dbtoolboxdemo` as the data source name and `tutorial database` as the description.

7 Select the database for this data source to use. For some drivers, you can skip this step. If you are unsure about skipping this step, consult your database administrator.

   a In the ODBC Microsoft Access Setup dialog box, click **Select**.



   b Specify the database that you want to use. For the `dbtoolboxdemo` data source, select `tutorial.mdb`.

   c If your database is on a system to which your PC is connected:

      i Click **Network**.

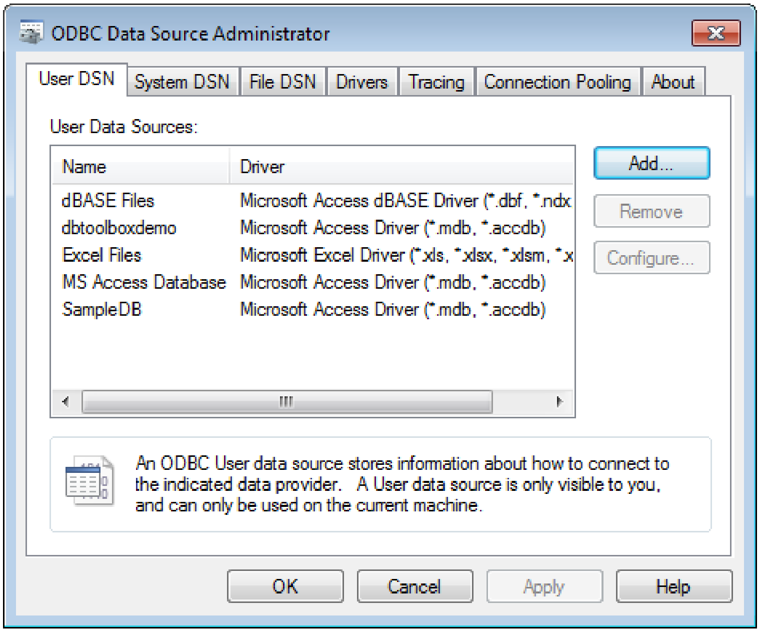
      ii In the Map Network Drive dialog box, specify the folder containing the database that you want to use. Click **Finish**.

   d Click **OK** to close the Select Database dialog box.

8 In the ODBC Microsoft Access Setup dialog box, click **OK**.

9 Repeat steps 6 through 8 with the following changes to define the data source for any additional databases that you want to use.

   The ODBC Data Source Administrator dialog box displays the `dbtoolboxdemo` and any additional data sources that you have added in the **User DSN** tab.



10 Click **OK** to close the dialog box.

Configure JDBC Data Sources

1 Find the name of the JDBC driver file. This file is provided by your database vendor. The name and location of this file differ for each system. If you do not know the name or location of this file, consult your database administrator.

Requirement: If you use Visual Query Builder (`querybuilder`) to access a JDBC data source, before starting Database Explorer for the first time, execute this command . You cannot use the same JDBC data source with Database Explorer.

```
setdbprefs('JDBCDataSourceFile','')
```

Then follow these instructions to set up the JDBC data source using Database Explorer.

---

2 Specify the location of the JDBC driver file in the MATLAB Java class path by adding the JDBC driver file path to the `javaclasspath.txt` file. At the start of each session, MATLAB loads the static class path. The static path offers better class loading performance than the dynamic path. To add folders to the static path, create the file `javaclasspath.txt`, and then restart MATLAB.

Create an ASCII file in your preferences folder. Name the file `javaclasspath.txt`. To view the location of the preferences folder, type:

```
prefdir
```

Each line in the file is the path of a folder or JAR file. For example:

```
d:\work\javaclasses
```

To simplify the specification of folders in cross-platform environments, use any of these macros: `$matlabroot`, `$arch`, or `$jre_home`.

Note: MATLAB reads the static class path only at startup. If you edit `javaclasspath.txt` or change your `.class` files while MATLAB is running, restart MATLAB to put those changes into effect.

If the drivers file is not located where `javaclasspath.txt` indicates, errors do not appear, and Database Explorer does not establish a database connection.

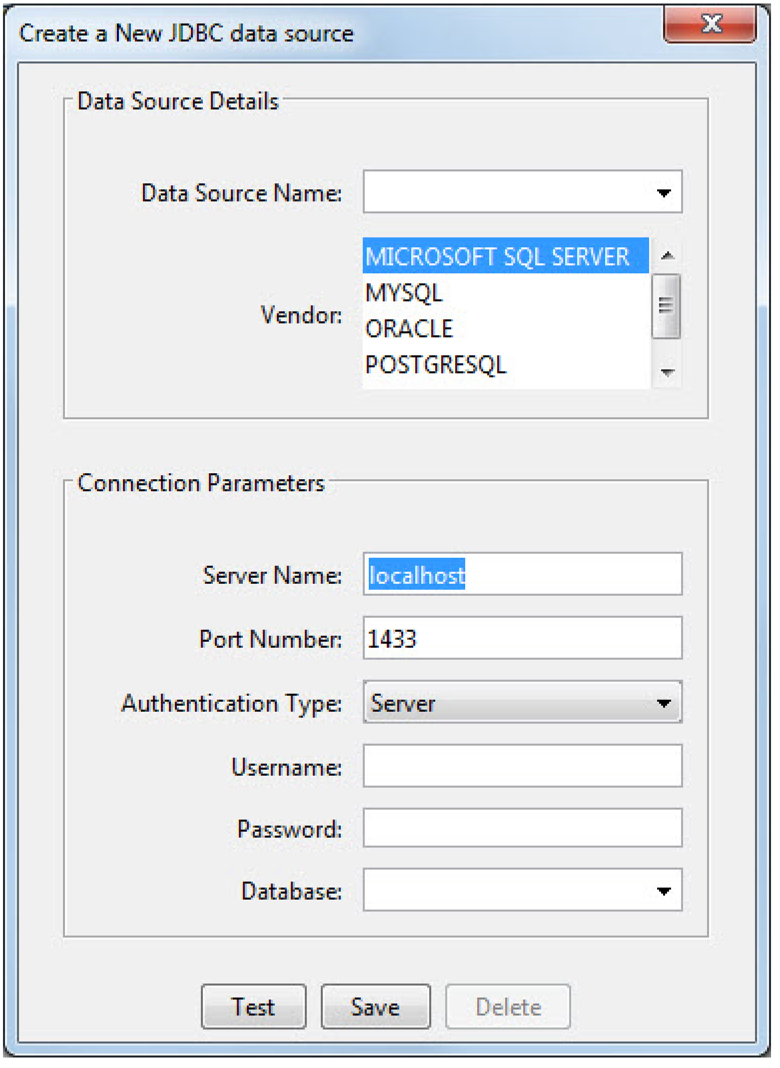
Alternatively, you can create a `javaclasspath.txt` file in your MATLAB startup folder. The classes that you specify in this file override the classes that you specify in the `javaclasspath.txt` file in the preferences folder.

For details, see “Bring Java Classes into MATLAB Workspace”.

3. Close the open database, `tutorial.mdb`, in the database program.
4. Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, at the command line, enter:

   ```
   dexplore
   ```

5. Click the **Database Explorer** tab. Select **New** > **JDBC** to open the Create a New JDBC data source dialog box.



6 Use the information in the following table to set up JDBC drivers for Database Explorer.

a Use the Create a New JDBC data source dialog box. This table describes the fields that you use to define your JDBC data source. For examples of syntax for these fields, see "JDBC Driver Name and Database Connection URL" on page 8-52.

Field	Description
Data Source Name	The name that you assign to the data source. For some databases, **Name** must match the name of the database as recognized by the machine that it runs on.
Vendor	The vendor name for the data source. When using **Other** as a vendor: • **Driver** — The JDBC driver name (sometimes referred to as the class that implements the Java SQL driver for your database). • **URL** — The JDBC URL object, of the form `jdbc:`*`subprotocol`*`:`*`subname`*. *`subprotocol`*, is a database type. *`subname`* can contain other information that the **Driver** uses, such as the location of the database or a port number. It can take the form `//`*`hostname`*`:`*`port`*`/`*`databasename`*. **Note:** When using **Other** as the **Vendor**, your database driver documentation specifies the **Driver** and **URL** formats. For help with this information, consult your database system administrator.
Server Name	Server name.
Port Number	Server port number.
Authentication Type	(Microsoft SQL Server only) Server or Windows authentication.
Driver Type	(Oracle only) Driver type is **thin** or **oci**.
Username	User name to access the database.
Password	Password.
Database	Database name.

b In the Create a New JDBC data source dialog box, click **Save**.

c. When you are creating a data source using Database Explorer for the first time, the New file to store JDBC connection parameters dialog box opens. Use this dialog box to create a MAT-file that saves your specified data source information for future Database Explorer sessions.

   Navigate to the folder where you want to put the MAT-file, specify a name for it that includes a `.mat` extension, and click **Save**.

d. To test the connection, click **Test**.

   If your database requires a user name and password, a dialog box opens prompting you to supply them. Enter values into these fields and click **OK**.

   A confirmation dialog box states that the database connection succeeded.

e. To add more data sources, repeat steps 5 and 6 for each new data source.

Note: You can use tabs in Database Explorer to access different data sources. All of the data sources that you create using Database Explorer are stored in a single MAT-file for easy access. This MAT-file name is stored in `setdbprefs('JDBCDataSourceFile')` and is valid for all MATLAB sessions.

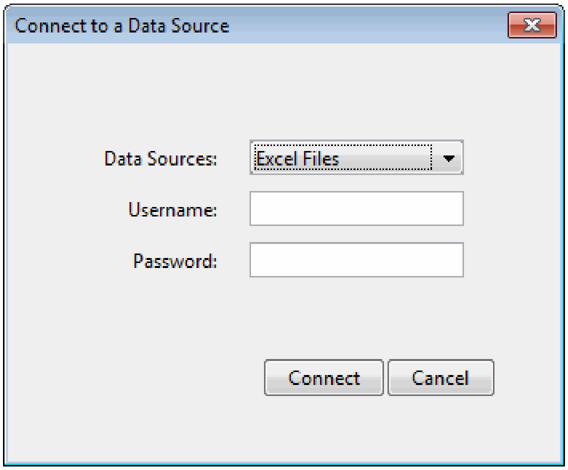
Connect to a Data Source

After configuring your OBDC or JDBC data sources, use Database Explorer to connect to the database.

1. Open Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip. Select **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, at the command line, enter:

   ```
   dexplore
   ```

2. In the Connect to a Data Source dialog box, select your data source. Or, click **Cancel**, click the **Database Explorer** tab, and click **Connect** to select your data source.
3. Select your data source from the **Data Sources** list. Enter your user name and password.



For details about potential errors, see “Database Connection Error Messages” on page 3-10.

Work with Multiple Databases

1. If you have not defined the OBDC or JDBC connection for your new data source, click **Open** and select **ODBC** or **JDBC**. Complete the fields in the associated dialog box. For details, see “Configure ODBC Data Sources” on page 4-8 or “Configure JDBC Data Sources” on page 4-12.
2. Select **Connect** > **Connect** to select your new data source.
3. The new data source appears in a new tab in the **Database Browser** pane. You can change databases by clicking the associated tab.

   You can use only Database Explorer to create SQL queries for a single database at a time.

You can work with a different catalog and schema on the same database server as the one that connects to your current data source. To change to a different catalog and schema:

- From the drop-down list in the address bar of the Database Browser, select the catalog or schema. For database systems that have a hierarchy of catalogs and schemas, ensure that you choose the correct value for catalogs and schemas to access data in your tables.

See Also

`database`

More About

- "Choosing Between ODBC and JDBC Drivers" on page 2-13
- "Configuring a Driver and Data Source" on page 2-16
- "Working with Database Explorer" on page 4-2

Modify and Delete Database Connections

ODBC Drivers

For data sources that you create with ODBC drivers, you can modify the data source using the ODBC Data Source Administrator. For details, see "Configuring a Driver and Data Source" on page 2-16.

1 Click **Start**. Select **Administrative Tools > Data Sources (ODBC)**. The ODBC Data Source Administrator dialog box opens. For details about locating ODBC Data Source Administrator on your computer, see Driver Installation.
2 In the ODBC Data Source Administrator dialog box, select the data source that you want to modify. Click **Configure**.
3 Modify the settings as needed.

For data sources that you create with ODBC drivers, you can delete the data source using the ODBC Data Source Administrator.

1 After opening the ODBC Data Source Administrator, select the data source that you want to delete.
2 Click **Remove**.

JDBC Drivers

For data sources that you create with JDBC drivers, you can modify the data source using Database Explorer. For details, see "Configuring a Driver and Data Source" on page 2-16.

1 Open Database Explorer and click the **Database Explorer** tab. Select **New** > **JDBC**.
2 Select the data source name that you want to modify from the drop-down list.
3 Modify the settings in the Create a New JDBC data source dialog box. If you leave the data source name as is, the data source name is overwritten with the new

settings. If you do not want to overwrite the existing data source, enter a new data source name. Click **Save**.

For data sources that you create with JDBC drivers, you can delete the data source using the Database Explorer.

1 After opening Database Explorer, select **New** > **JDBC**.
2 Select the data source name that you want to delete from the drop-down list. Click **Delete**.

Refine Results Using Query Criteria and Rules

In this section...

To define a query without writing SQL code, use the **SQL Criteria** panel in Database Explorer. Define criteria to build your query within one table or join multiple tables in the database. Build your query using the **SQL Criteria** panel rules and control options.

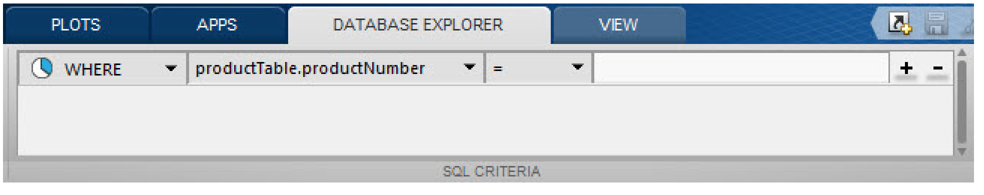
Define Query Criteria to Refine Results

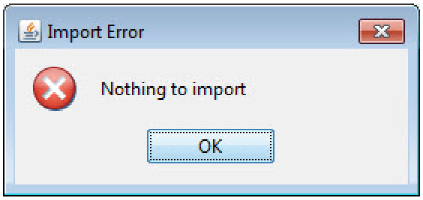
Database Browser selections and SQL criteria work together.

Using the **Database Browser** pane and the **SQL Criteria** panel, you can define query conditions and display the results in the **Data Preview** pane. Each row in the **SQL Criteria** panel has a drop-down list of controls to define SQL query conditions. You can create SQL query conditions that span multiple rows in the **SQL Criteria** panel.

Requirement: When you enter a custom value in the text box on the right side of a query condition, press **Enter** or **Tab** to apply it. Alternatively, you can click the **Import** button to apply the condition and import data into a MATLAB variable.

If you do not use the **Enter** or **Tab** keys to apply the query condition, selecting **Import** > **Import** applies the condition to the **Data Preview** pane. Then, Database Explorer imports the data into a MATLAB variable. If there is no data to satisfy the condition, then the Nothing to import error message appears.





Each row in the **SQL Criteria** panel has four columns to define your SQL query.

Column 1	Column 2	Column 3	Column 4
Column 1 defines the SQL condition type where the supported values are: • **INNER JOIN** • **LEFT JOIN** • **RIGHT JOIN** • **FULL JOIN** • **WHERE** • **ORDER BY** • **AND** • **OR**	Column 2 defines the column names for every table that you select in the **Database Browser** pane.	Column 3 defines the mathematical operator for each row of SQL statements where the supported values are: • **=** • **!=** • **>** • **<** • **<=** • **>=** • **LIKE** • **NOT LIKE** • **IS** • **IN** • **NOT IN** • **ASC** • **DES**	Depending on the preceding condition of the query statement, Column 4 displays column names for every table that you select in the **Database Browser** pane.

Use multiple rows in the **SQL Criteria** panel to define multiple SQL query statements.

Query Rules Using the SQL Criteria Panel

The control options for the **SQL Criteria** panel depend on your selections in the **Database Browser** pane. The **SQL Criteria** panel supports multiple rows for specifying your query criteria. You can add more rows for these options in the **SQL Criteria** panel by clicking **+**. You can remove a row by clicking **-**.

- If you select one table in the **Database Browser** pane, the available options for the first query condition are **WHERE** and **ORDER BY**.
- If you select two tables in the **Database Browser** pane, the available options for the first query condition are:
  - **INNER JOIN**
  - **LEFT JOIN**
  - **RIGHT JOIN**
  - **FULL JOIN**
  - **WHERE**
  - **ORDER BY**
  - **AND**
  - **OR**
- Press **Enter** or **Tab** to apply a condition for a row. The first (leftmost) column contains query options that produce semantically correct SQL statements for each subsequent condition that you add. For example, the leftmost column of an applied condition contains an **ORDER BY** option. If you click **+** to add a query option in a new row, the **ORDER BY** option can follow only another **ORDER BY**.

  A **Join** option can follow only another **JOIN** or **WHERE**. A **JOIN** option cannot follow a **WHERE** or **ORDER BY** option.
- When defining a query line for any conditions other than a **JOIN**, the line does not take effect until you apply it. When you apply a condition, the software removes all preceding and succeeding conditions that you did not apply from the **SQL Criteria** panel. Similarly, if you click **-** to remove a query line, if you have applied that query line, all succeeding conditions are removed. If you have not yet applied the query line, the software removes only that line from the **SQL Criteria** panel.
- When using a **WHERE** SQL statement with a mathematical operator, to match a string, include the string value in `'  '` to apply the condition. If you use the **LIKE** or **NOT LIKE** SQL operator to match a string, the software adds the `'  '` to the string value.

Note: If you click **+** to add a query condition between two previously entered conditions, the available query options do not always produce semantically correct SQL statements. In this case, ensure that your query options are semantically correct. For best results using the **SQL Criteria** panel, add and apply your conditions in sequence.

More About

- "Working with Database Explorer" on page 4-2

Generate SQL and MATLAB Code

In this section...

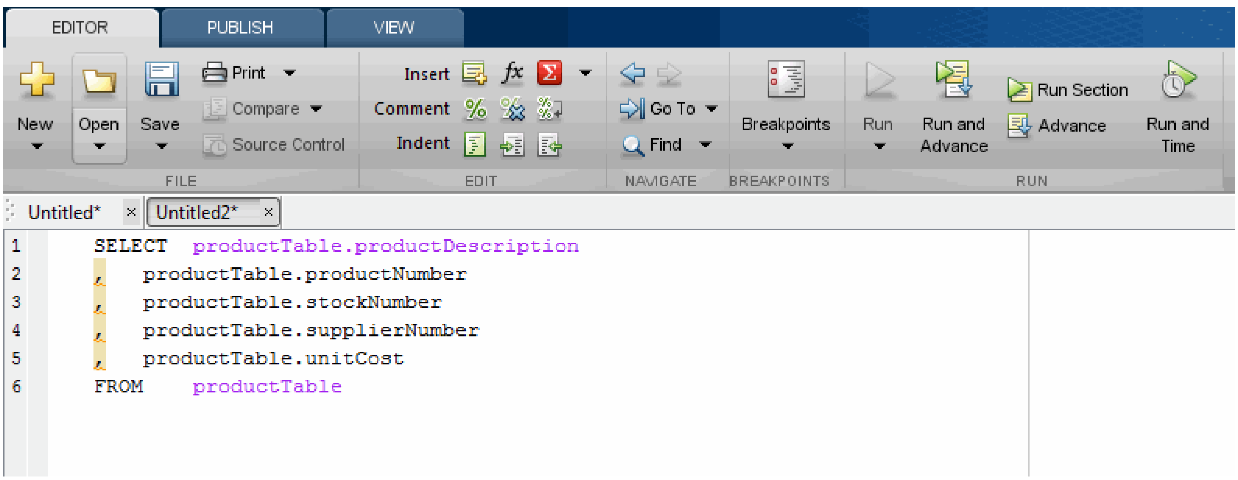
Use Database Explorer to generate SQL or MATLAB code. Once you define a SQL query using the **SQL Criteria** panel, you can generate the SQL code for running a SQL script. You can also generate MATLAB code to automate connecting to the database, running a SQL query, and performing data analysis on the imported data.

Save Queries as SQL Code

You can save a Database Explorer query as SQL code.

1 In the **Database Browser** pane, select data from a single table or multiple tables. Use the **SQL Criteria** panel to create queries and display the results in the **Data Preview** pane.

2 After you have created a query using the **SQL Criteria** panel, select **Import** > **Generate SQL** to display the SQL code in the MATLAB Editor.

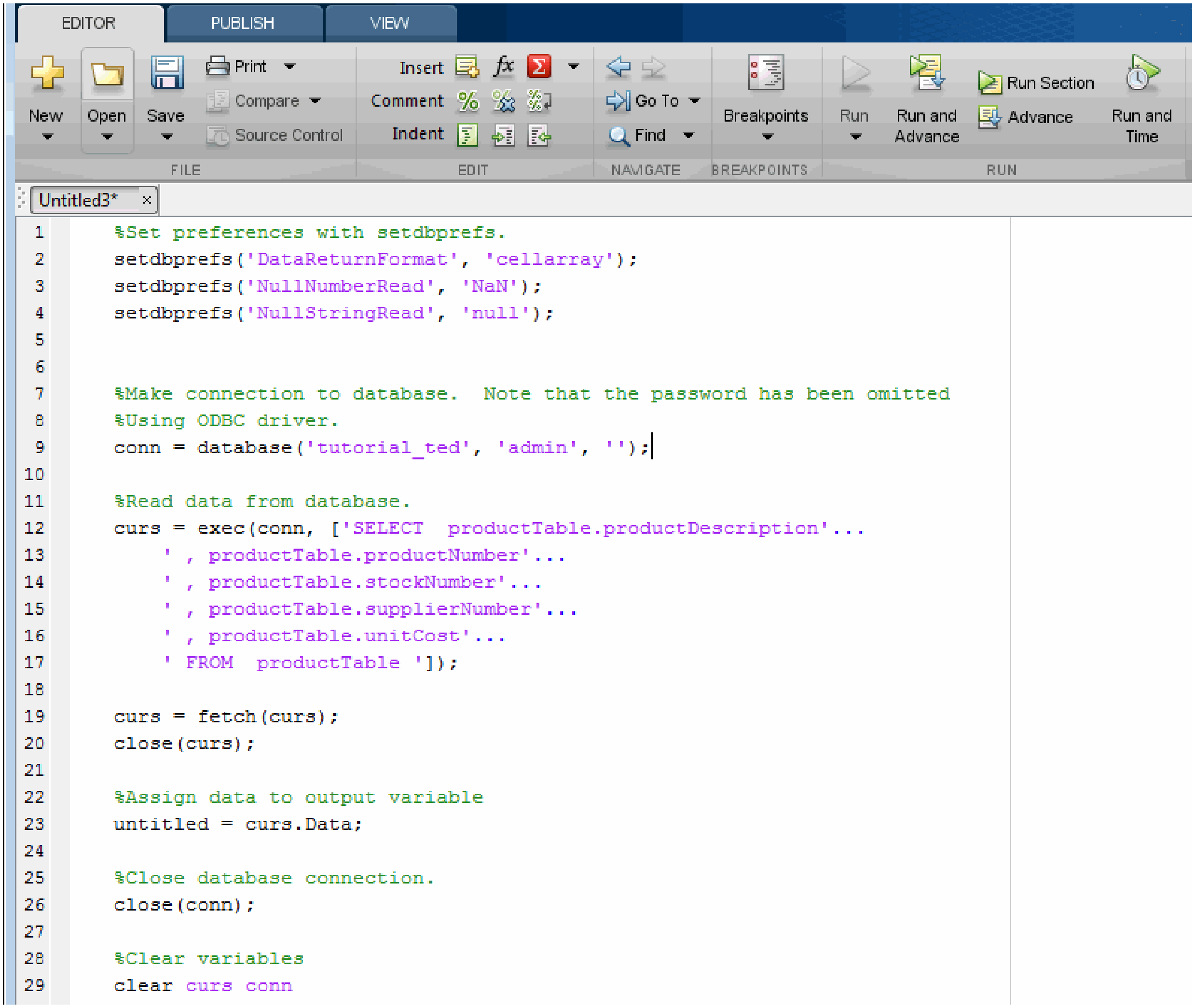


3 Save the SQL code to a `.txt` or `.sql` file. You can then use the SQL statements to rebuild a query using the **SQL Criteria** panel. Alternatively, you can use the `.sql` file to import data programmatically into MATLAB by using `runsqlscript`.

Generate MATLAB Code

You can generate MATLAB code to automate accessing data that you display in the **Data Preview** pane.

1 Connect to a data source. Use the **SQL Criteria** panel to create a query and display the results in the **Data Preview** pane.
2 Select **Import** > **Generate Script** to display MATLAB code in the MATLAB Editor.



3 Save the MATLAB code to a file. You can run this code file at the command line to connect to a data source and run a query.

See Also

`runsqlscript`

More About

- "Working with Database Explorer" on page 4-2

5

Using Visual Query Builder

Getting Started with Visual Query Builder

In this section...

What Is Visual Query Builder?

Visual Query Builder (VQB) is an easy-to-use graphical user interface (GUI) for exchanging data with your database. To start VQB, use `querybuilder`. You can use VQB to:

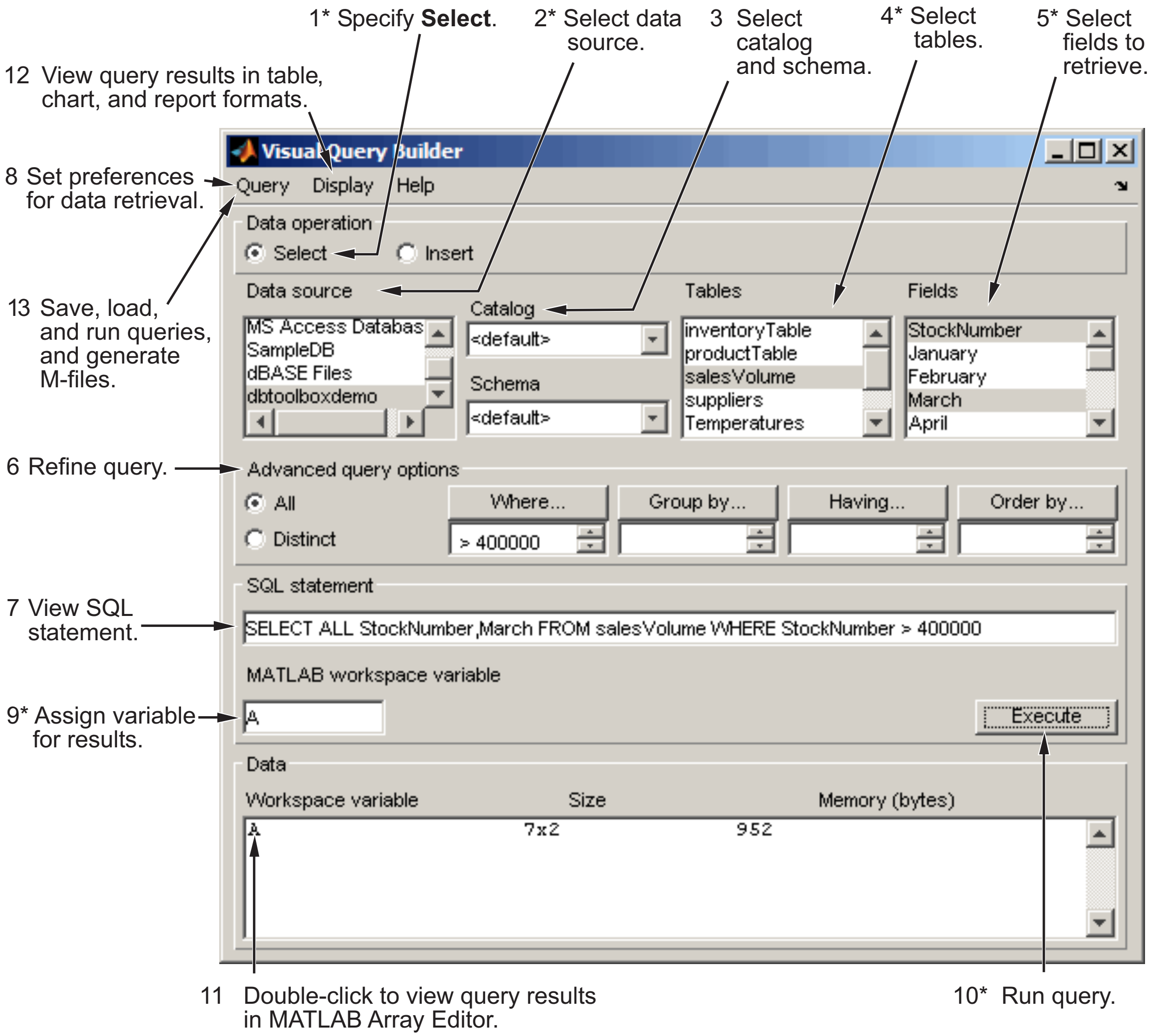
- Build queries to retrieve data by selecting information from lists instead of using MATLAB functions.
- Store data retrieved from a database in a MATLAB cell array, structure, or numeric matrix.
- Process the retrieved data using the MATLAB suite of functions.
- Display retrieved information in relational tables, reports, and charts.
- Export data from the MATLAB workspace into new rows in a database.

Using Queries to Import Data

The following steps summarize how to use VQB to import data.

To start the Visual Query Builder, type `querybuilder` at the MATLAB prompt.

*Required step



You can graphically construct and run SQL queries to import database data using:

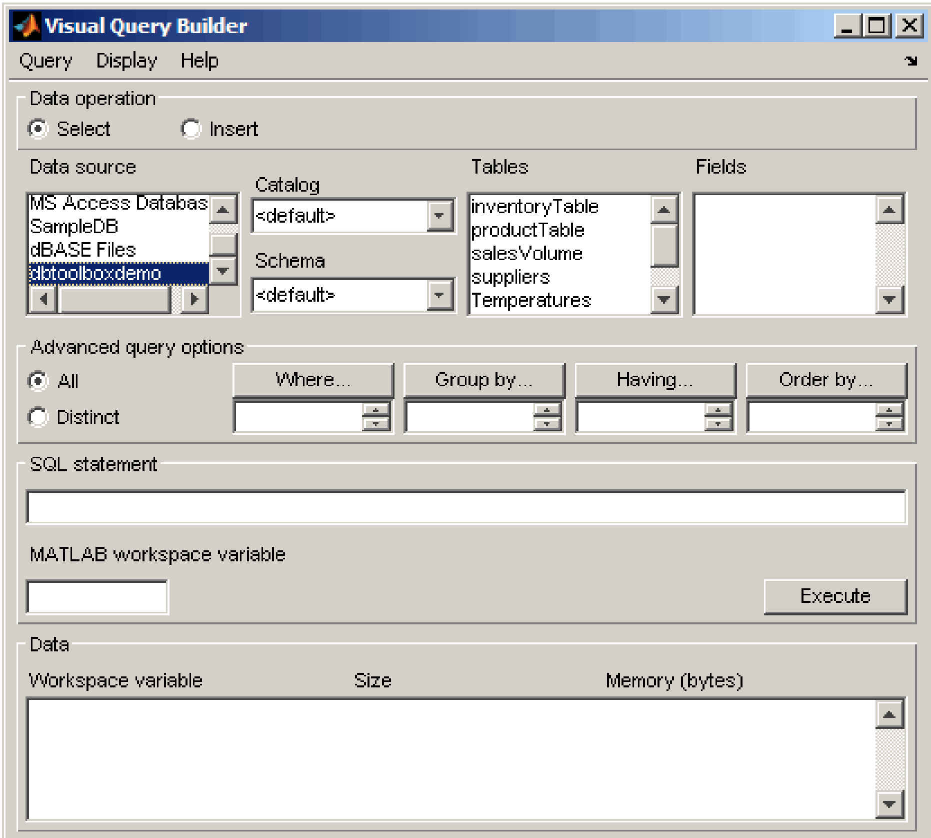
- Visual Query Builder (`querybuilder`)
- Database Explorer

To create and run a query using Visual Query Builder to import data from a database into the MATLAB workspace:

1 Select data from a database by clicking the **Select** button under **Data operation**. The data sources that you defined in “Configuring a Driver and Data Source” on page 2-16 appear.

2 Select `dbtoolboxdemo` as the data source from which to import data.

   After you select a data source, the catalog, schema, and tables for your specified data source appear in the **Catalog**, **Schema**, and **Tables** fields.



3 Accept the default values `<default>` for the **Catalog** and **Schema** fields. Setting these fields to the default values indicates that you have not specified a catalog or schema.

Tip To populate the VQB **Schema** and **Catalog** fields, you must associate your user name with schemas or catalogs before starting VQB.

- To specify a **Catalog**, select one from the list, and then select a schema from within that catalog. The **Schema** field updates to reflect your selections.

- Alternatively, you can select a schema without specifying a catalog; that is, when the **Catalog** field set to `<default>`. The **Tables** field updates to reflect the schema you selected.

4 In the **Tables** list, select `salesVolume` as the table that contains the data you want to import.

The set of **Fields** (column names) in the table appears.

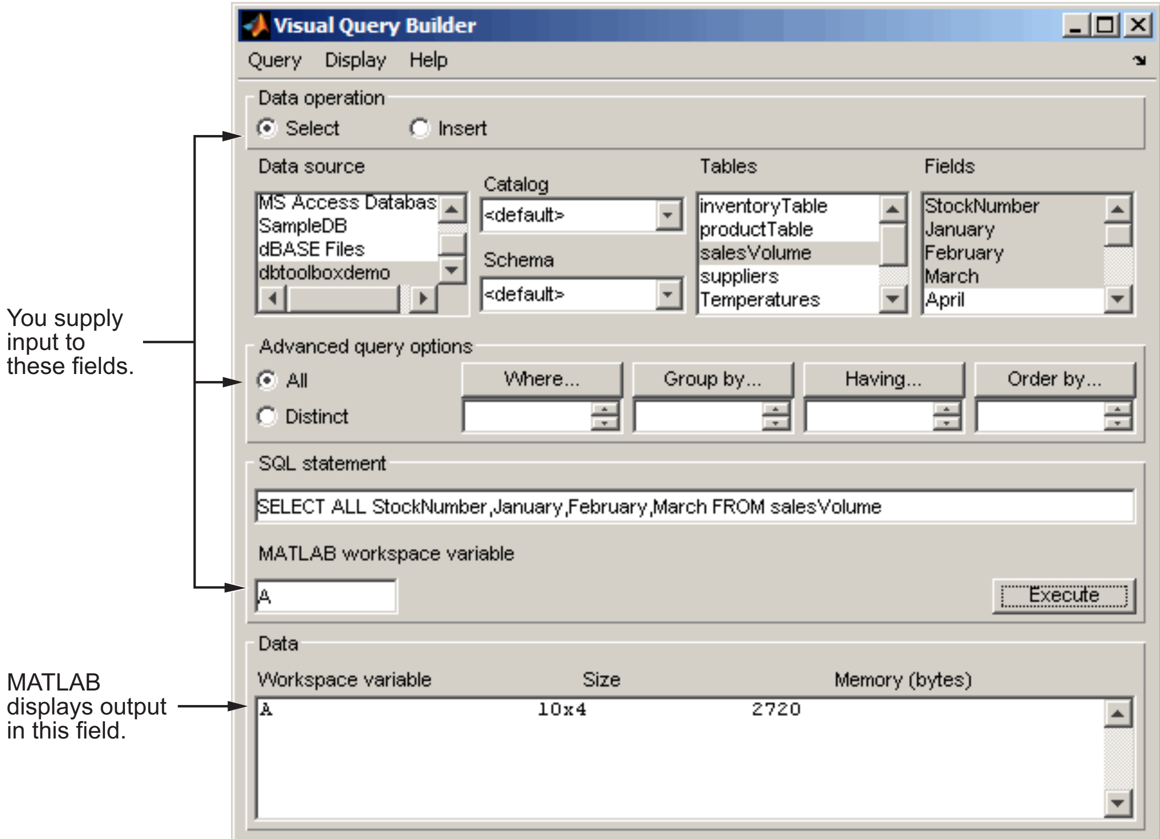
5 In the **Fields** list, select `StockNumber`, `January`, `February`, and `March` as the fields that contain the data you want to import.

Tip To select more than one field, hold down the **Ctrl** or **Shift** key while selecting multiple fields. To clear an entry, use **Ctrl**+click.

VQB adds each field you select to the query in the **SQL statement** field.

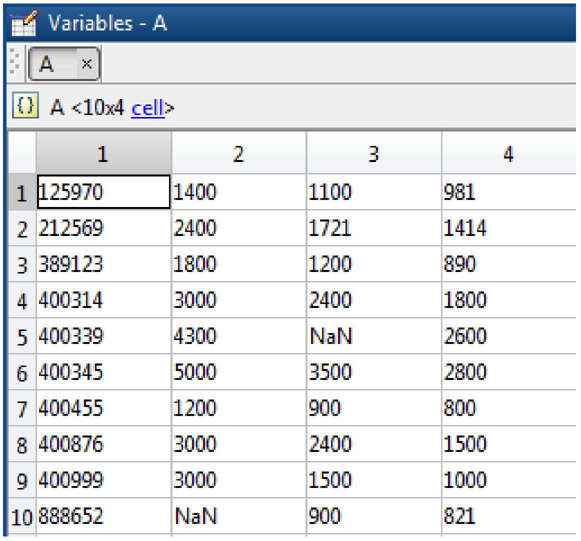
6 Enter the name `A` in the **MATLAB workspace variable** field. `A` is a cell array that stores the data that the query returns.

7 Click **Execute** to run the query and import the data. The **Data** field displays information about the query result.



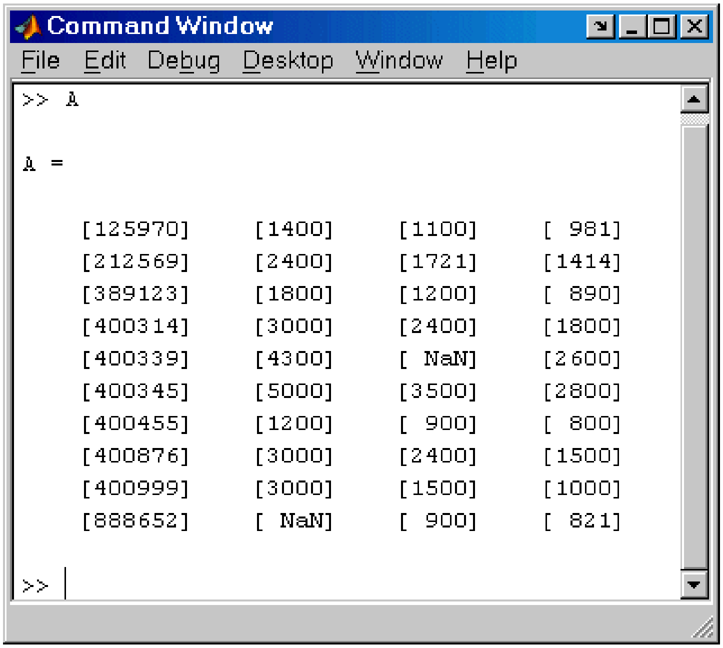
8 Double-click A in the **Data** area. The contents of A appear in the Variables editor, where you can view and edit the data. In this example, sales for item 400876 are 3000 in January, 2400 in February, and 1500 in March.

For details about using the Variables editor, see "Create and Edit Variables".



	1	2	3	4
1	125970	1400	1100	981
2	212569	2400	1721	1414
3	389123	1800	1200	890
4	400314	3000	2400	1800
5	400339	4300	NaN	2600
6	400345	5000	3500	2800
7	400455	1200	900	800
8	400876	3000	2400	1500
9	400999	3000	1500	1000
10	888652	NaN	900	821

Alternatively, you can view the contents of A by entering A in the Command Window.

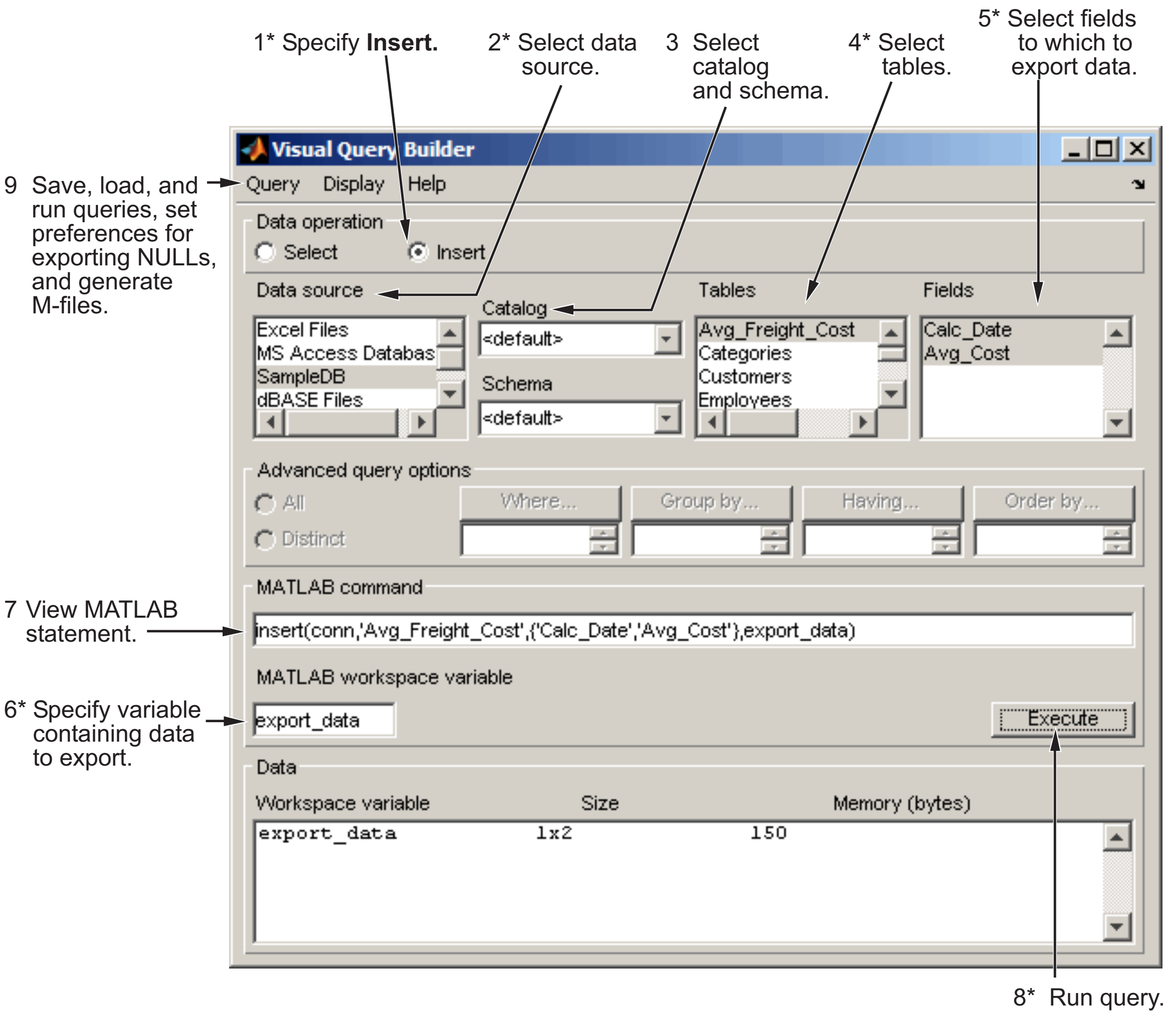


Using Queries to Export Data

The following steps summarize how to use VQB to export data.

To start the Visual Query Builder, type `querybuilder` at the MATLAB prompt.

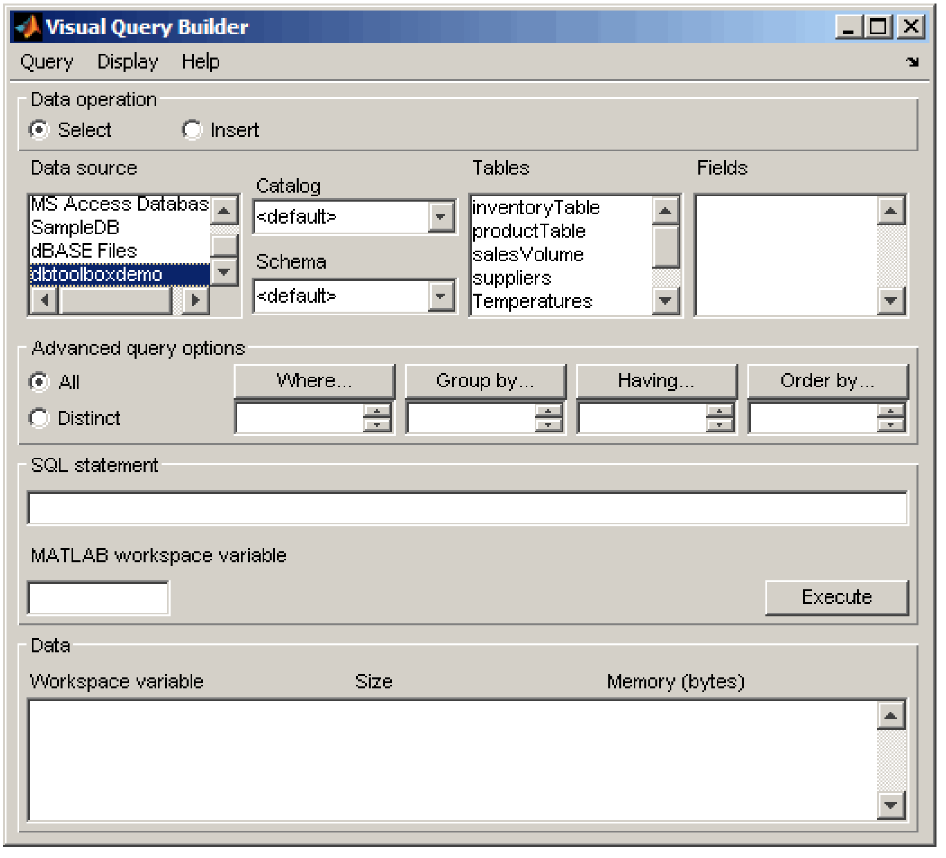
*Required step



To build, run, and save a query that exports data from the MATLAB workspace into new rows in a database:

1 Select **Data Operation** > **Insert** to select data to export.

2 Select `dbtoolboxdemo` as the data source to which to export data from the **Data source** list box. The **Catalog**, **Schema**, and **Tables** fields for `dbtoolboxdemo` appear.



3 Do not specify values for **Catalog** and **Schema**.

4 In the **Tables** list box, select `inventoryTable` as the table to which you want to export data from the MATLAB software.

The set of **Fields** (column names) in your selected table appears.

5 In the **Fields** list box, select `productNumber`, `Quantity`, and `Price` as the fields to which you want to export data from the MATLAB software.

VQB adds each field you select to the query in the **MATLAB command** field.

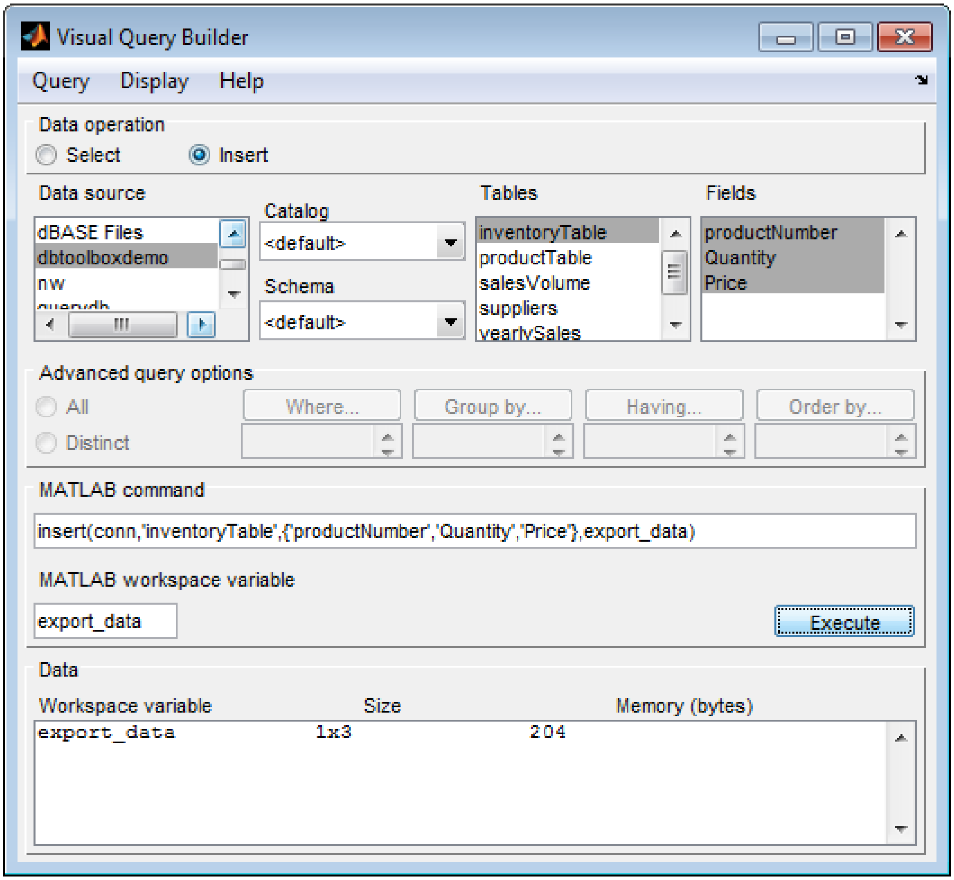
6 In the MATLAB workspace, assign the data you want to export to a cell array, `export_data`.

```
export_data = {14,1500,18.50};
```

7 In the **MATLAB workspace variable** field, enter the name of the variable containing data to export, `export_data`. Press **Enter** or **Return** to view the **MATLAB command** that exports the data.

8 Click **Execute** to run the query to export the data.

Information about the exported data appears in the **Data** area.



9 View the `inventoryTable` table in the Microsoft Access database to verify the query results.

inventoryTable

productNum	Quantity	Price
1	1700	$14.50
2	1200	$9.00
3	356	$17.00
4	2580	$21.00
5	9000	$3.00
6	4540	$8.00
7	6034	$16.00
8	8350	$5.00
9	2339	$13.00
10	723	$24.00
11	567	$0.00
12	1278	$0.00
13	1700	$14.50
14	1500	$18.50

10 To save this query, select **Query** > **Save** and name it `export.qry`.

Clearing Variables from the VQB Data Area

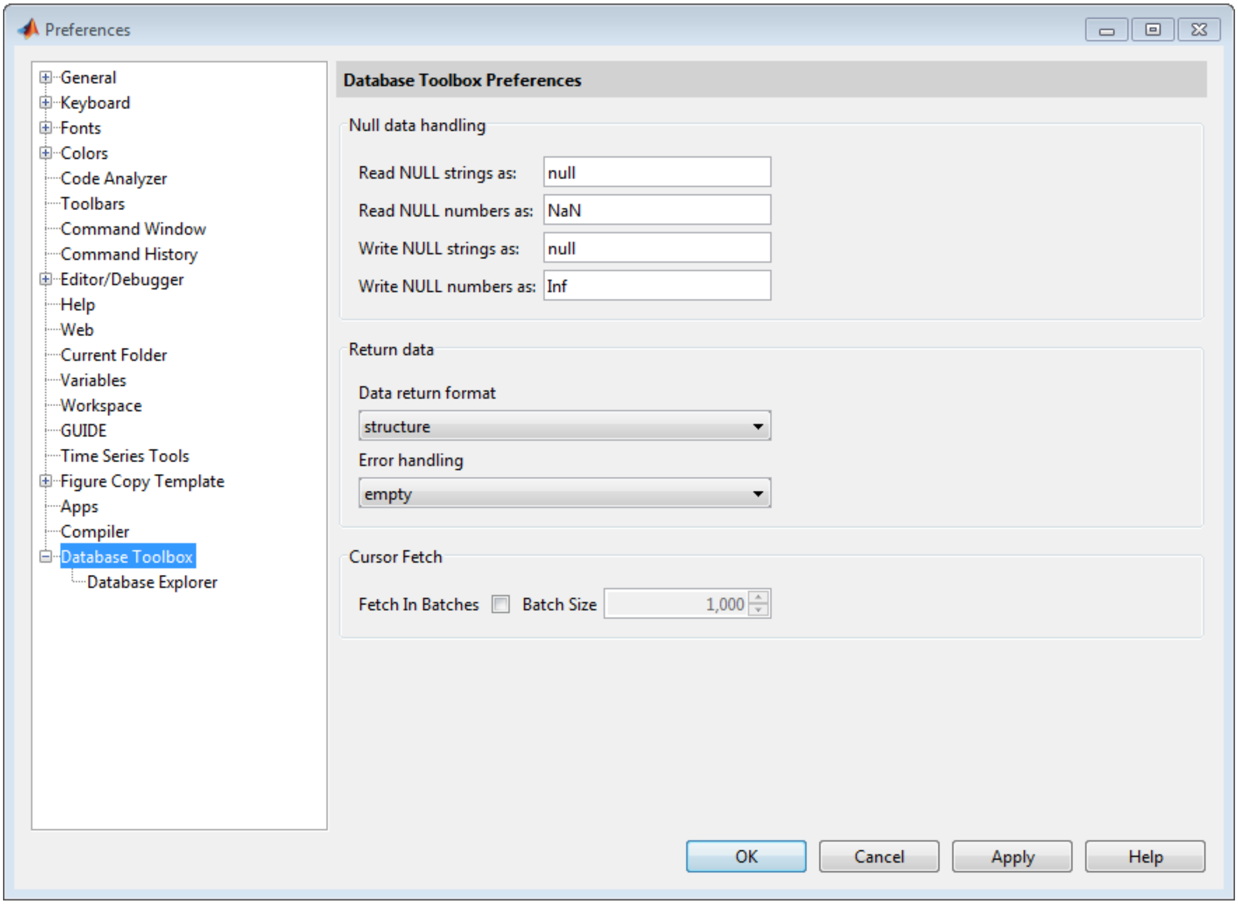
Variables in the **Data** area include those you assigned in the Command Window and those that contain query results. The variables do not appear in the **Data** area until you execute a query. They then remain in the **Data** area until you clear them. To clear the variables, run the `clear` function in the Command Window.

Working with Preferences

Database Toolbox preferences enable you to specify:

- How `NULL` data in a database is represented after you import it into the MATLAB workspace
- The format of data retrieved from databases
- The method of error notification
- The preference for fetching in batches

1 From Visual Query Builder, select **Query** > **Preferences**. The Preferences dialog box appears. Alternatively, from the MATLAB Toolstrip, click **Preferences** and select **Database Toolbox**.



2 Specify the Preferences settings as described in this table.

Preference	Acceptable Values	Description
Read NULL strings as:	`null` (default)	Specifies how `NULL` strings appear after being fetched from a database.
Read NULL	`Nan` (default)	Specifies how `NULL` numbers appear after being fetched from a database. If you accept the default value for this field, `NULL` data imported from databases into the MATLAB workspace appears

Preference	Acceptable Values	Description
numbers as:		as `NaN`. Setting this field to 0 causes `NULL` data imported into the MATLAB workspace to appear as `0`s.
Write NULL strings as:	`null` (default)	Specifies how `NULL` strings appear after being exported to a database. This setting does not apply to Database Explorer.
Write NULL numbers as:	`Nan` (default)	Specifies how `NULL` numbers appear after being exported to a database. This setting does not apply to Database Explorer.
Data return format	`cell array`, `numeric`, `structure`, or `dataset`	Select a data format based on the type of data you are importing, memory considerations, and your preferred method of working with retrieved data. • `cellarray` (default) — Imports nonnumeric data into MATLAB cell arrays. • `numeric` — Imports data into MATLAB matrix of doubles. Nonnumeric data types are considered `NULL` and appear as specified in the **Read NULL numbers as:** setting. Use only when data to retrieve is in numeric format, or when nonnumeric data to retrieve is not relevant. • `structure` — Imports data into a MATLAB structure. Use for all data types. Facilitates working with returned columns. • `dataset` — Imports data into MATLAB `dataset` objects. This option requires Statistics and Machine Learning Toolbox™. This setting does not apply to Database Explorer. If you are using Database Explorer, the data return format is specified using the **Imported Data** panel in Database Explorer.
Error handling	`store`, `report`, or `empty`	• Set this field to `store` or `empty` to direct errors to either a dialog box when using Visual Query Builder or a message field when using the Database Toolbox command line interface. • Set this field to `report` to display query errors in the Command Window. This setting does not apply to Database Explorer.

Preference	Acceptable Values	Description
Cursor Fetch	`Fetch In Batches` and `Batch Size`	Specifies if `fetch` retrieves data in batches with a user-defined `Batch Size`. The default `Batch Size` is 1,000. For details, see “Preference Settings for Large Data Import” on page 5-19. This setting does not apply to Database Explorer. If you are using Database Explorer, the import batch size is specified using **Preferences** on the Database Explorer Toolstrip.

3 Click **OK**. For details about Preferences, see `setdbprefs`.

Preference Settings for Large Data Import

In this section...

When using the `setdbprefs` to set `'FetchInBatches'` and `'FetchBatchSize'` or the **Cursor Fetch** option for the Preference dialog box, use the following guidelines to determine what batch size value to use. These guidelines are based on evaluating:

- The size of your data (n rows) to import into MATLAB
- The JVM heap requirements for the imported data

The general logic for making these evaluations are:

- If your data (n rows) will fit in a MATLAB variable, then will all your data fit in the JVM heap?
  - If yes, use the following preference setting:

    ```
    setdbprefs('FetchInBatches','no')
    ```
  - If no, evaluate `h` such that `h` < n and data of size `h` rows fits in the JVM heap. Use the following preference setting:

    ```
    setdbprefs('FetchInBatches','yes')
    setdbprefs('FetchBatchSize','h')
    ```
- If your data (n rows) will not fit in a MATLAB variable, then:
  - Evaluate `m` such that `m` < n and the data of size `m` rows fits in a MATLAB variable.
  - Evaluate `h` such that `h` < `m` < n and data of size `h` rows fits in the JVM heap. Use the following preference setting:

    ```
    setdbprefs('FetchInBatches','yes')
    setdbprefs('FetchBatchSize','h')
    ```
    Then import data using `fetch` or `runsqlscript` by using the value `'m'` to limit the number of rows in the output:

    ```
    curs = fetch(curs,m)
    ```

    or

```
results = runsqlscript(conn,'filename.sql','rowInc','m')
```

- If you are using the native ODBC interface to import large amounts of data, you do not need to change these settings because the native ODBC interface always fetches data in batches of 100,000 rows. You can still override the default batch size by setting `'FetchInBatches'` to `'yes'` and `'FetchBatchSize'` to a number of your choice. Note that JVM heap memory restrictions do not apply in this case since the native ODBC interface is a C++ API.

Will All Data (Size n) Fit in a MATLAB Variable?

This example shows how to estimate the size of data to import from a database.

It is important to have an idea of the size of data that you are looking to import from a database. Finding the size of the table(s) in the database can be misleading because MATLAB representation of the same data is most likely going to consume more memory. For instance, say your table has one numeric column and one text column and you are looking to import it in a cell array. Here is how you can estimate the total size.

```
data = {1001, 'some text here'};
whos data
```

```
 Name      Size            Bytes  Class    Attributes

  data      1x2               156  cell
```

If you are looking to import a thousand rows of the table, the approximate size in MATLAB would be 156 * 1000 = 156 KB. You can replicate this process for a structure or a dataset array depending on which data type you want to import the data in. Once you know the size of data to be imported in MATLAB, you can determine whether it fits in a MATLAB variable by executing the command `memory` in MATLAB.

A conservative approach is recommended here so as to take into account memory consumed by other MATLAB tasks and other processes running on your machine. For example, even if you have 12 GB RAM and the memory command in MATLAB shows 14 GB of longest array possible, it might still be a good idea to limit your imported data to a reasonable 2 or 3 GB to be able to process it without issues. Note that these numbers vary from site to site.

Will All of This Data Fit in the JVM Heap?

This example shows how to determine the size of the JVM heap.

The value of your JVM heap can be determined by selecting MATLAB **Preferences** and **General** > **Java Heap Memory**. You can increase this value to an allowable size, but keep in mind that increasing JVM heap reduces the total memory available to MATLAB arrays. Instead, consider fetching data in small batches to keep a low to medium value for heap memory.

How Do I Perform Batching?

There are three different methods based on your evaluations of the data size and the JVM heap size. Let n be the total number of rows in the data you are looking to import, `m` be the number of rows that fit in a MATLAB variable, and `h` be the number of rows that fit in the JVM heap.

Method 1 — Data Does Not Fit in MATLAB Variable or JVM Heap

If your data (n) does not fit in a MATLAB variable or a JVM heap, you need to find `h` and `m` such that `h` < `m` < n.

To use automated batching to fetch those `m` rows in MATLAB:

```
setdbprefs('FetchInBatches','yes')
setdbprefs('FetchBatchSize','h')
```

If using `exec`, `fetch`, and connection object `conn`:

```
curs = exec(conn,'Select….');
curs = fetch(curs,m);
```

If using `runsqlscript` to run a query from an SQL file `'filename.sql'`:

```
results = runsqlscript(conn,'filename.sql','rowInc','m')
```

Once you are done processing these `m` rows, you can import the next `m` rows using the same commands. Keep in mind, however, that using the same cursor object `curs` for this results in the first `curs` being overwritten, including everything in `curs.Data`.

Note: If `'FetchInBatches'` is set to `'yes'` and the total number of rows fetched is less than `'FetchBatchSize'`, MATLAB shows a warning message and then fetches all the rows. The message is: Batch size specified was larger than the number of rows fetched.

Method 2 — Data Does Fit In MATLAB Variable But Not in JVM Heap

If your data (n) does fit in a MATLAB variable but not in a JVM heap, you need to find `h` such that `h` < n.

To use automated batching to fetch where `h` rows fit in the JVM heap:

```
setdbprefs('FetchInBatches','yes')
setdbprefs('FetchBatchSize','h')
```

If using `exec`, `fetch`, and the connection object `conn`:

```
curs = exec(conn,'Select….');
curs = fetch(curs);
```

If using `runsqlscript` to run a query from an SQL file `'filename.sql'`:

```
results = runsqlscript(conn,'filename.sql')
```

Note that when you use automated batching and do not supply the `rowLimit` parameter to `fetch` or the `rowInc` parameter to `runsqlscript`, a count query is executed internally to get the total number of rows to be imported. This is done to preallocate the output variable for better performance. In most cases, the count query overhead is not much, but you can easily avoid it if you know or have a good idea of the value of `n`:

```
curs = fetch(curs,n)
```

or

```
results = runsqlscript(conn,'filename.sql','rowInc','n')
```

Method 3 — Data Fits in MATLAB Variable and JVM Heap

If your data (n) fits in a MATLAB variable and also in a JVM heap, then you need not use batching at all.

```
setdbprefs('FetchInBatches','no')
```

If using `fetch`:

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

If using `runsqlscript` to run a query from an SQL file `'filename.sql'`:

```
results = runsqlscript(conn,'filename.sql')
```

See Also

`exec` | `fetch` | `runsqlscript` | `setdbprefs`

More About

- "Working with Large Data Sets" on page 2-199

Displaying Query Results

In this section...

How to Display Query Results

To display query results, perform one of the following actions:

- Enter the variable name to which to assign the query results in the Command Window.
- Double-click the variable in the VQB **Data** area to view the data in the Variables editor.

The examples in this section use the saved query `basic.qry`. To load and configure this query:

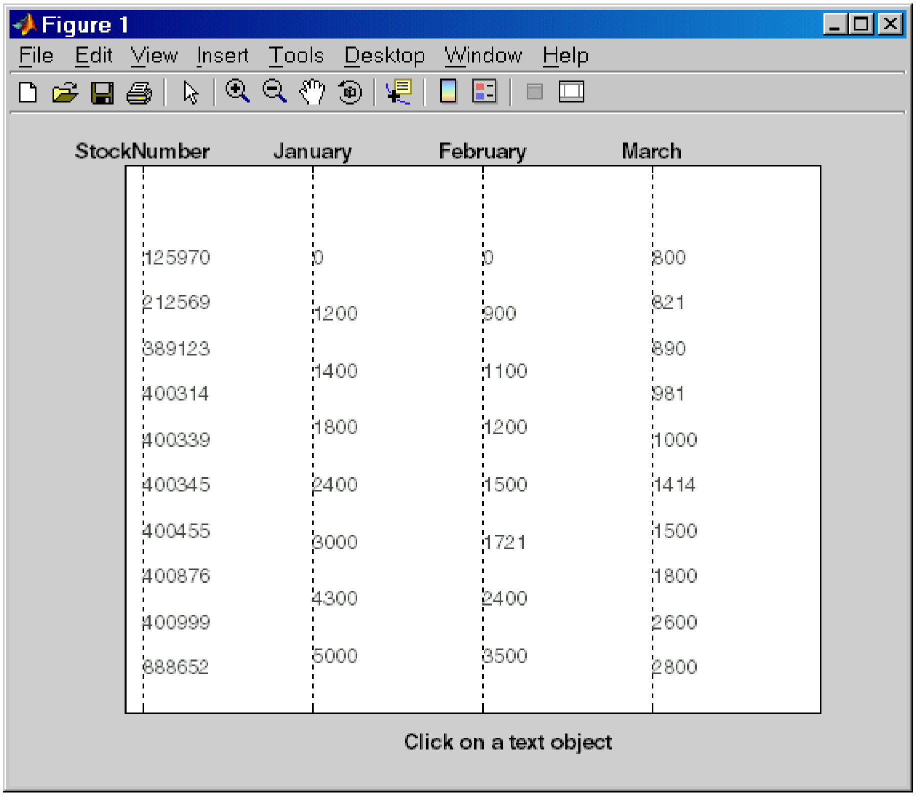
1. Select **Query** > **Preferences**, and set **Read NULL numbers as** to `0`.
2. Select **Query** > **Load**.
3. In the Load SQL Statement dialog box, select `basic.qry` from the **File name** field and click **Open**.
4. In VQB, enter a value for the **MATLAB workspace variable**, for example, `A`, and click **Execute**.

Displaying Data Relationally

To display the results of `basic.qry`:

1. Execute `basic.qry`.
2. Select **Display** > **Data**.

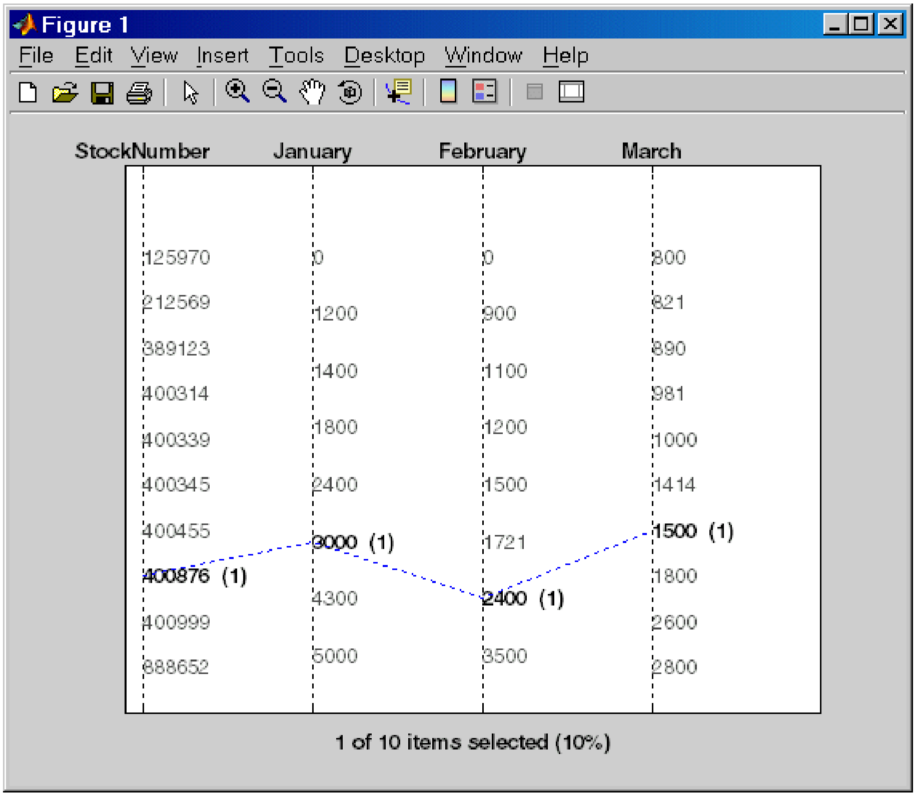
   The query results appear in a figure window.



This display shows only unique values for each field, so you should not read each row as a single record. In this example, there are 10 entries for **StockNumber**, eight entries for **January** and **February**, and 10 entries for **March**. The number of entries in each field corresponds to the number of unique values in the field.

3. Click a value in the figure window, for example, **StockNumber** `400876`, to see its associated values.

   The data associated with the selected value appears in bold font and is connected with a dotted line. The data shows that sales for item 400876 are 3000 in January, 2400 in February, and 1500 in March.



4 As another example, click 3000 under **January**. It shows three different items with sales of 3000 units in January: 400314, 400876, and 400999.

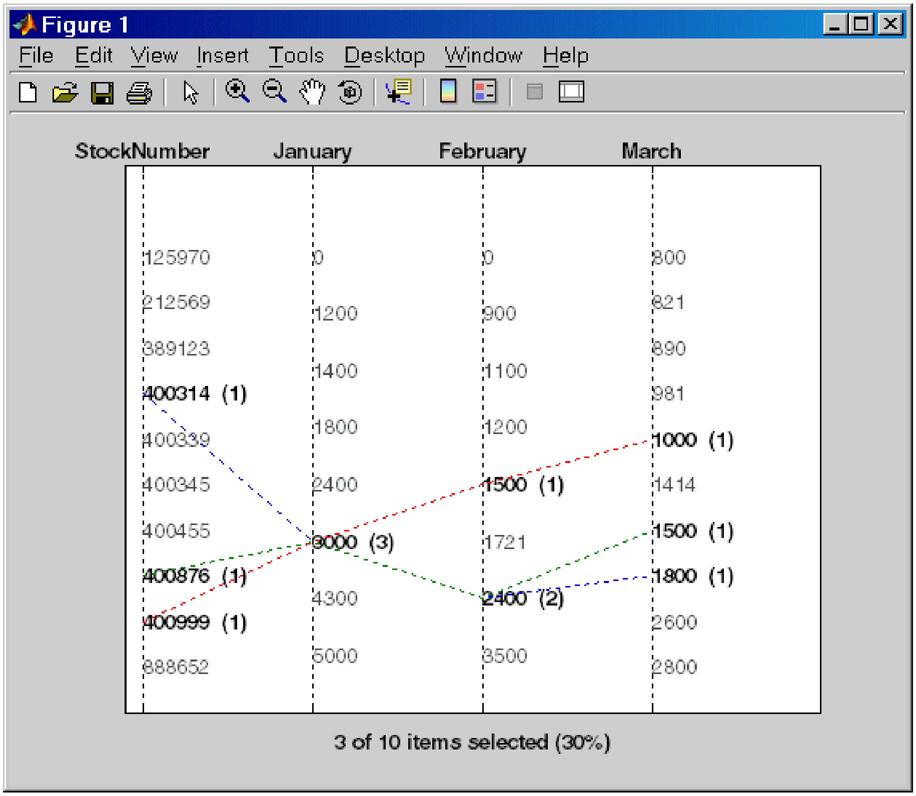


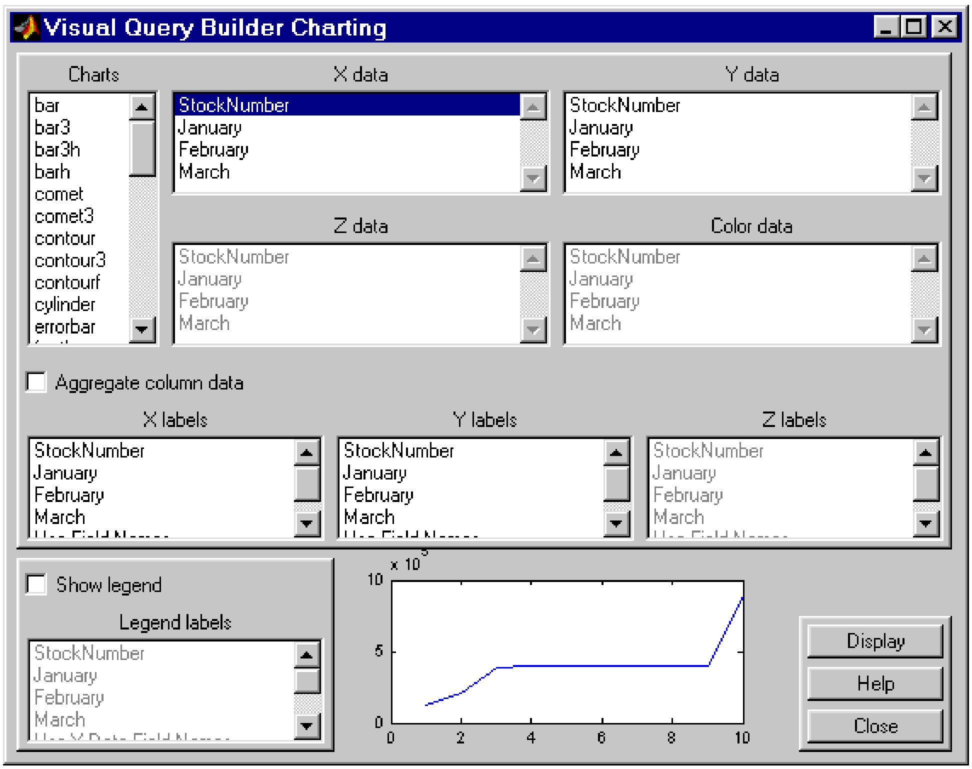
Figure 1
File Edit View Insert Tools Desktop Window Help
StockNumber
January
February
March
125970
212569
389123
400314 (1)
400339
400345
400455
400876 (1)
400999 (1)
888652
0
1200
1400
1800
2400
3000 (3)
4300
5000
0
900
1100
1200
1500 (1)
1721
2400 (2)
3500
800
821
890
981
1000 (1)
1414
1500 (1)
1800 (1)
2600
2800
3 of 10 items selected (30%)

Charting Query Results

To chart the results of `basic.qry`:

1 Select **Display** > **Chart**.

The Visual Query Builder Charting dialog box appears.



2 Select a type of chart from the **Charts** list. In this example, choose a pie chart by specifying `pie`.

A preview of the pie chart, with each stock item displayed in a different color, appears at the bottom of the dialog box.

3 Select the data to display in the chart from the **X data**, **Y data**, and **Z data** list boxes. In this example, select `March` from the **X data** list box to display a pie chart of March data.

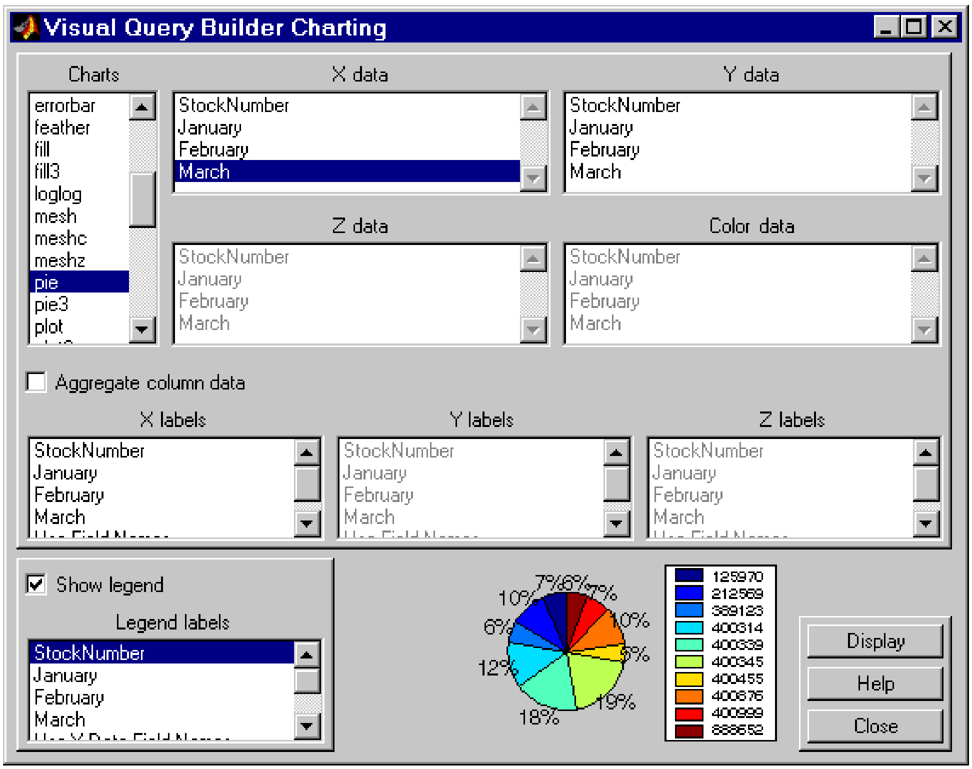
The pie chart preview now shows percentages for March data.

4 To display a legend, which maps colors to the stock numbers, select the **Show legend** check box.

The **Legend labels** field becomes active.

5 Select `StockNumber` from the **Legend labels** list box.

A legend appears in the chart preview. Drag and move the legend in the preview as needed.



6 Click **Close** to close the Charting dialog box.

Displaying Query Results in an HTML Report

To display results for `basic.qry` in an HTML report, select **Display** > **Report**.

The query results appear as a table in a Web browser. Each row represents a record from the database. In this example, sales for item 400876 are 3000 in January, 2400 in February, and 1500 in March.

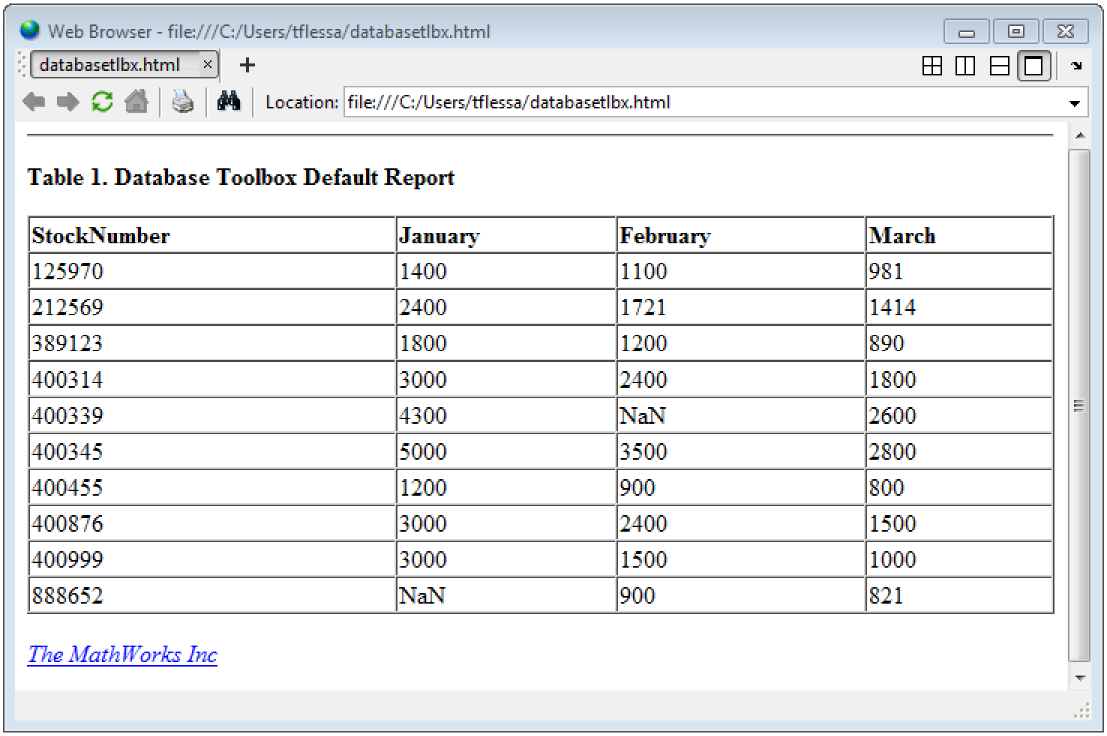


Table 1. Database Toolbox Default Report

StockNumber	January	February	March
125970	1400	1100	981
212569	2400	1721	1414
389123	1800	1200	890
400314	3000	2400	1800
400339	4300	NaN	2600
400345	5000	3500	2800
400455	1200	900	800
400876	3000	2400	1500
400999	3000	1500	1000
888652	NaN	900	821

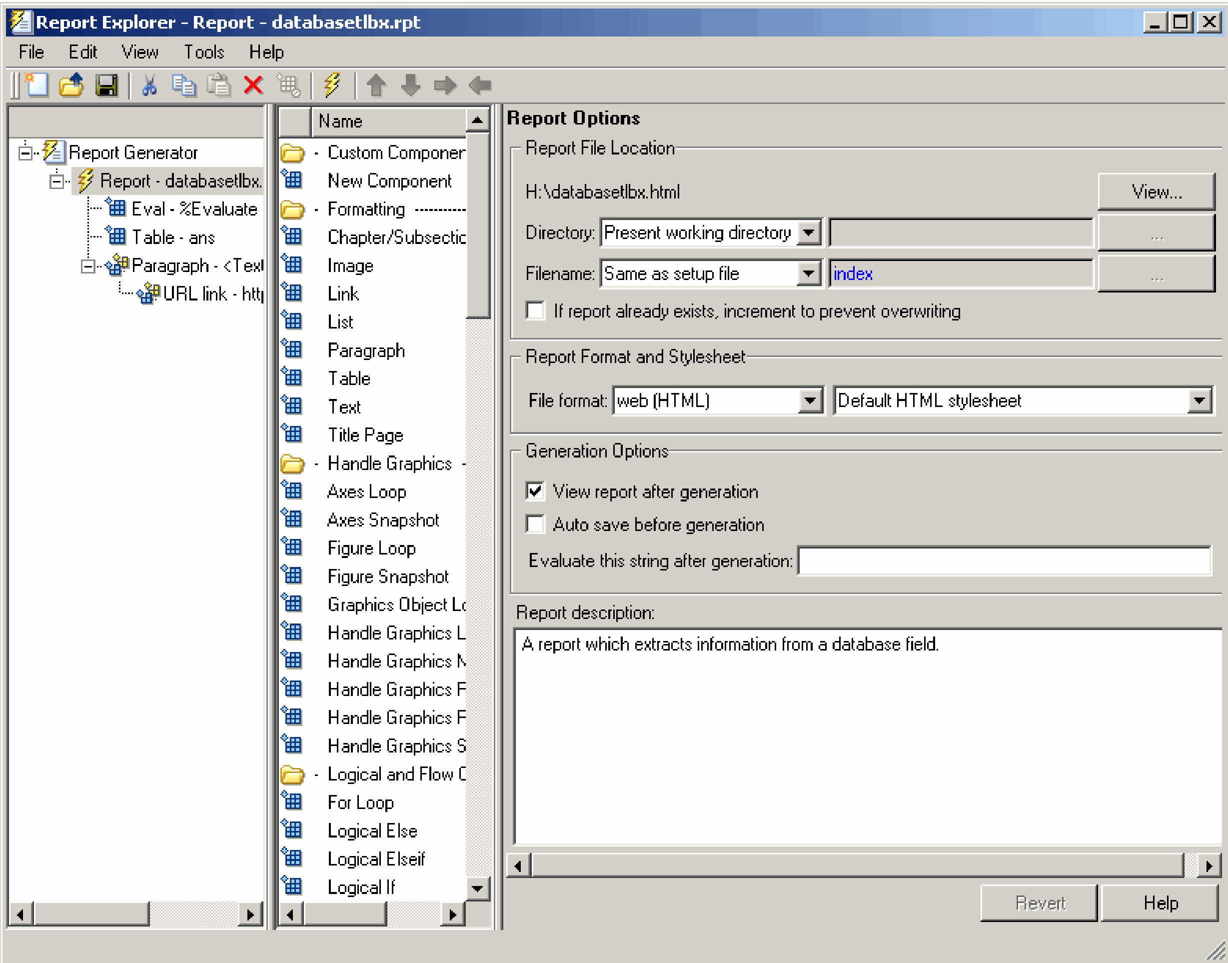
Tip Because some browsers do not start automatically, you may need to open your Web browser before displaying the query results.

Displaying Query Results with MATLAB Report Generator

To use the MATLAB Report Generator software to customize the display of the results of `basic.qry`:

1 Select **Display** > **Report Generator**.

2 The Report Explorer opens, listing sample report templates that you can use to create custom reports. Select the template `matlabroot/toolbox/database/vqb/databasetlbx.rpt` from the Options pane in the middle of the Report Explorer window.



3 Open the report template for editing by clicking **Open a Report file or stylesheet**.

a In the Outline pane on the left, under **Report Generator > databasetlbx.rpt**, select `Table`.

b In the Properties pane on the right, do the following:

i In **Table Content** > **Workspace Variable Name**, enter the name of the variable to which you assigned the query results in VQB, for example, `'A'`.

ii Under **Header/Footer Options**, set **Number of header rows** to `0`.

c Click **Apply**.

4 Select **File** > **Report** to run the report.

The report appears in a Web browser.

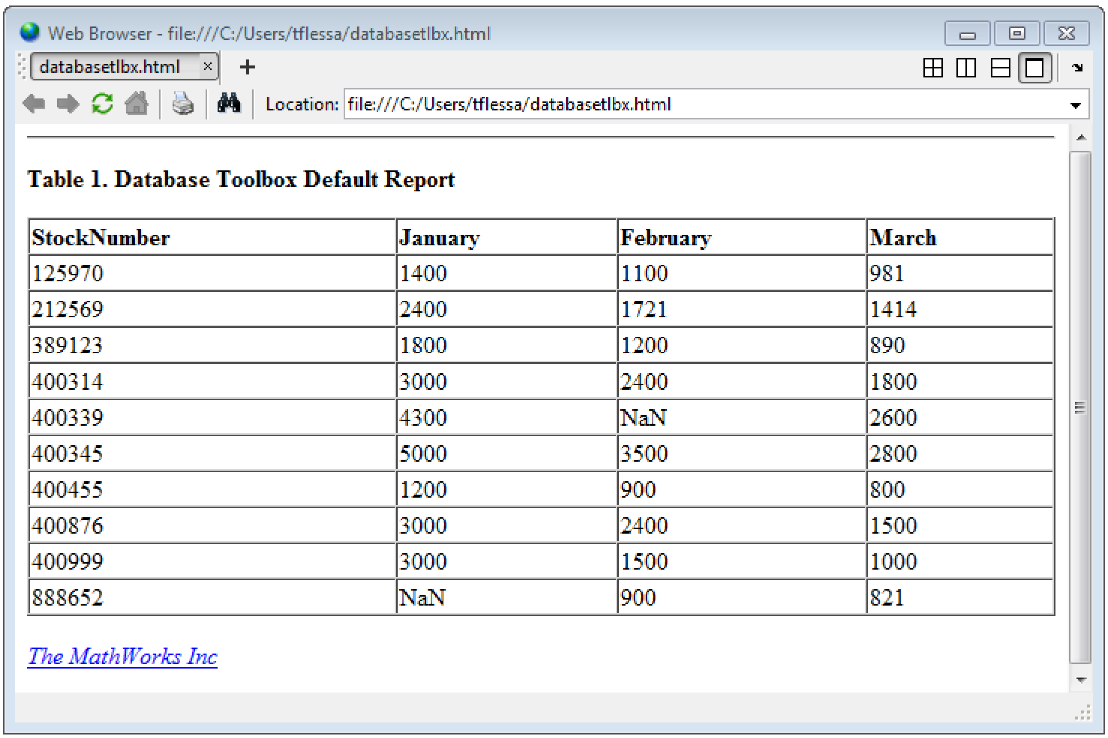


Table 1. Database Toolbox Default Report

StockNumber	January	February	March
125970	1400	1100	981
212569	2400	1721	1414
389123	1800	1200	890
400314	3000	2400	1800
400339	4300	NaN	2600
400345	5000	3500	2800
400455	1200	900	800
400876	3000	2400	1500
400999	3000	1500	1000
888652	NaN	900	821

5 Field names do not automatically display as column headers in the report. To display the field names:

a Modify the workspace variable A as follows:

```
A = [{'Stock Number', 'January', 'February', 'March'};A]
```

b In the MATLAB Report Generator Properties pane, change **Number of header rows** to 1 and regenerate the report. The report now displays field names as headings.

Each row represents a record from the database. For example, sales for item 400876 are 3000 in January, 2400 in February, and 1500 in March.

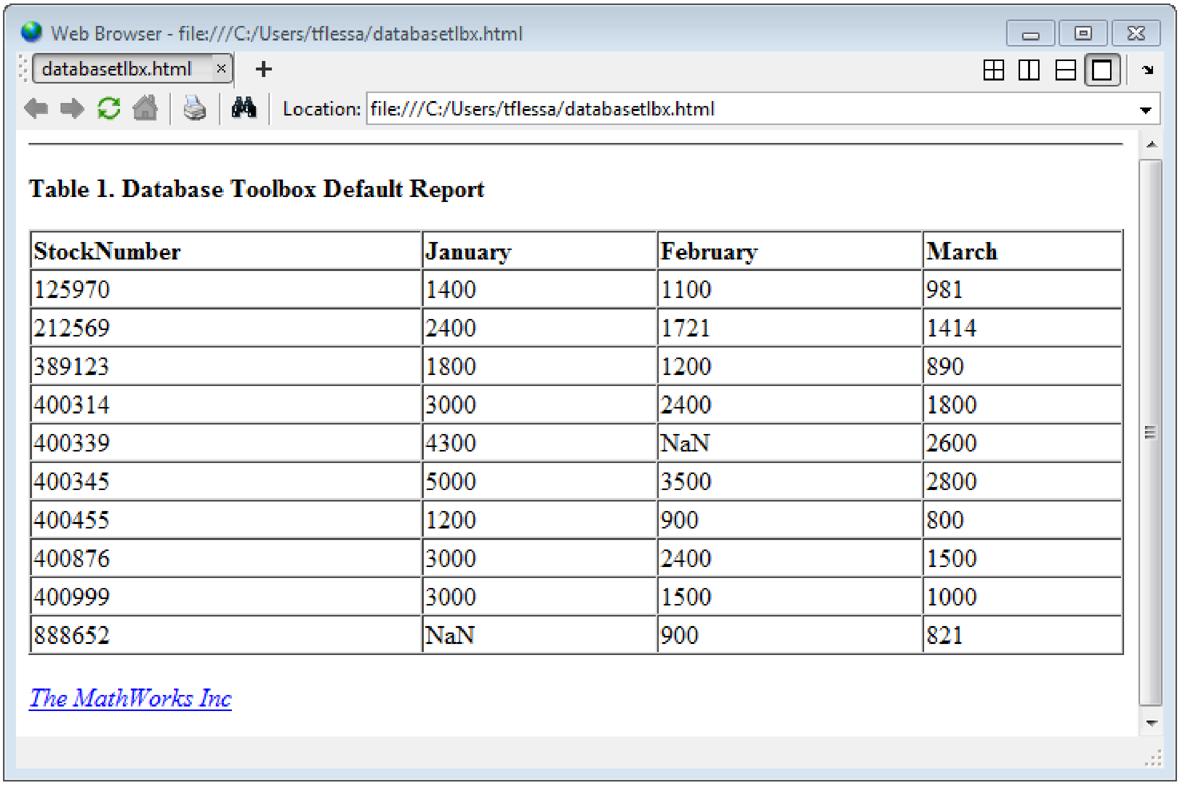


Table 1. Database Toolbox Default Report

StockNumber	January	February	March
125970	1400	1100	981
212569	2400	1721	1414
389123	1800	1200	890
400314	3000	2400	1800
400339	4300	NaN	2600
400345	5000	3500	2800
400455	1200	900	800
400876	3000	2400	1500
400999	3000	1500	1000
888652	NaN	900	821

The MathWorks Inc

For details about the MATLAB Report Generator product, click the **Help** button in the Report Explorer.

Tip Because some browsers are not configured to launch automatically, you may need to open your Web browser before displaying the report.

Fine-Tuning Queries Using Advanced Query Options

In this section...

Note: For details about advanced query options, select **Help** in any of the dialog boxes for the options.

Retrieving All Occurrences vs. Unique Occurrences of Data

To use the `dbtoolboxdemo` data source to demonstrate how to retrieve all versus distinct occurrences of data:

1. Set the **Data return format** preference to `cellarray`.
2. Set **Read NULL numbers as** to `NaN`.
3. In **Data operation**, choose **Select**.
4. In **Data source**, select `dbtoolboxdemo`.

   Do not specify **Catalog** or **Schema**.
5. In **Tables**, select `SalesVolume`.
6. In **Fields**, select `January`.
7. To retrieve all occurrences of `January`:

   a. In **Advanced query options**, select **All**.
   b. Assign the query results to the **MATLAB workspace variable** `All`.
   c. Click **Execute** to run the query.

8 To retrieve only unique occurrences of data:

   a In **Advanced query options**, select **Distinct**.

   b Assign the query results to a **MATLAB workspace variable** `Distinct`.

   c Click **Execute** to run the query.

9 In the Command Window, enter `All, Distinct` to display the query results:

```
All =

    [1400]
    [2400]
    [1800]
    [3000]
    [4300]
    [5000]
    [1200]
    [3000]
    [3000]
    [ NaN]


Distinct =

    [ NaN]
    [1200]
    [1400]
    [1800]
    [2400]
    [3000]
    [4300]
    [5000]
```

The value `3000` appears three times in `All`, but appears only once in `Distinct`.

Retrieving Data That Meets Specified Criteria

Use `basic.qry` and the **Where** field in **Advanced query options** to retrieve stock numbers greater than 400000 and less than 500000:

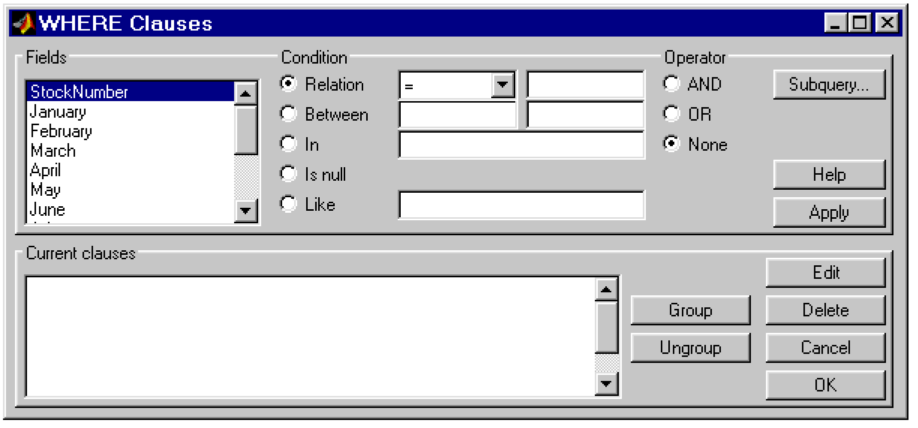
1 Load `basic.qry`.

2 Set the **Data return format** preference to `cellarray`.

3 Set **Read NULL numbers as** to `NaN`.

4 In **Advanced query options**, click **Where**.

The WHERE Clauses dialog box appears.



5 In **Fields**, select the field whose values you want to restrict, `StockNumber`.

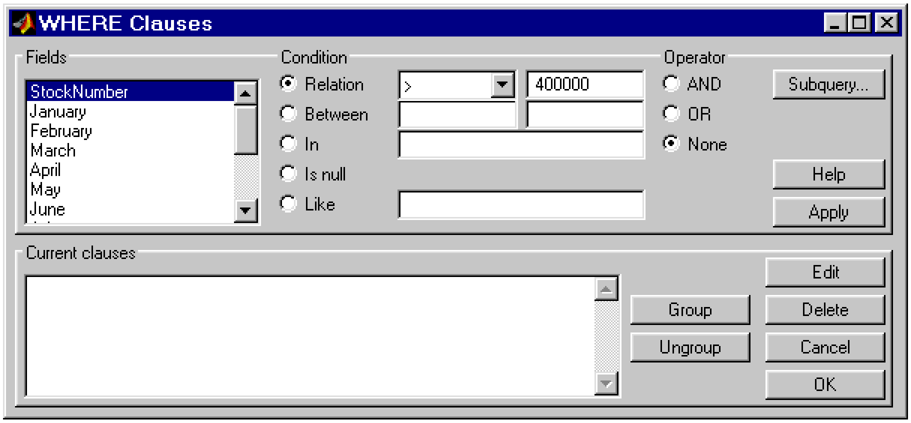
6 In **Condition**, specify that `StockNumber` must be greater than `400000`.

a Select **Relation**.

b In the drop-down list to the right of **Relation**, select `>`.

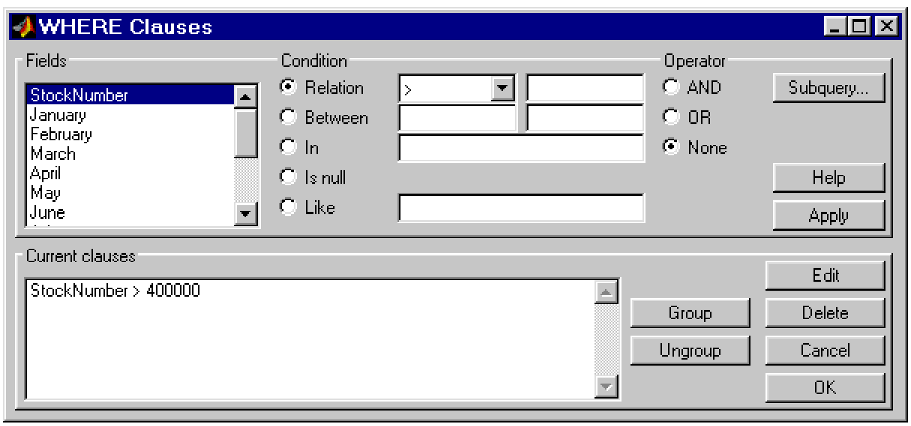
c In the field to the right of the drop-down list, enter `400000`.

The WHERE Clauses dialog box now looks as follows.



d Click **Apply**.

The clause that you defined, `StockNumber > 400000`, appears in the **Current clauses** area.



7 Add the condition that `StockNumber` must also be less than `500000`.

a In **Current clauses**, select `StockNumber > 400000`.

b In **Current clauses**, click **Edit** or double-click the `StockNumber` entry.

c For **Operator**, select **AND**.

d Click **Apply**.

The **Current clauses** field now displays:

```
StockNumber > 400000 AND
```

e In **Fields**, select `StockNumber`.

f In **Condition**, select **Relation**.

g In the drop-down list to the right of **Relation**, select `<`.

h In the field to the right of the drop-down list, enter `500000`.

i Click **Apply**.

The **Current clauses** field now displays:

```
StockNumber > 400000 AND
StockNumber < 500000
```

8 Click **OK**.

The WHERE Clauses dialog box closes. The **Where** field and **SQL statement** display the Where Clause you specified.

9 Assign the query results to the **MATLAB workspace variable** `A`.

10 Click **Execute**.

11 To view the results, enter `A` in the Command Window:

```
A =

    [400314]    [3000]    [2400]    [1800]
    [400339]    [4300]    [ NaN]    [2600]
    [400345]    [5000]    [3500]    [2800]
    [400455]    [1200]    [ 900]    [ 800]
    [400876]    [3000]    [2400]    [1500]
    [400999]    [3000]    [1500]    [1000]
```

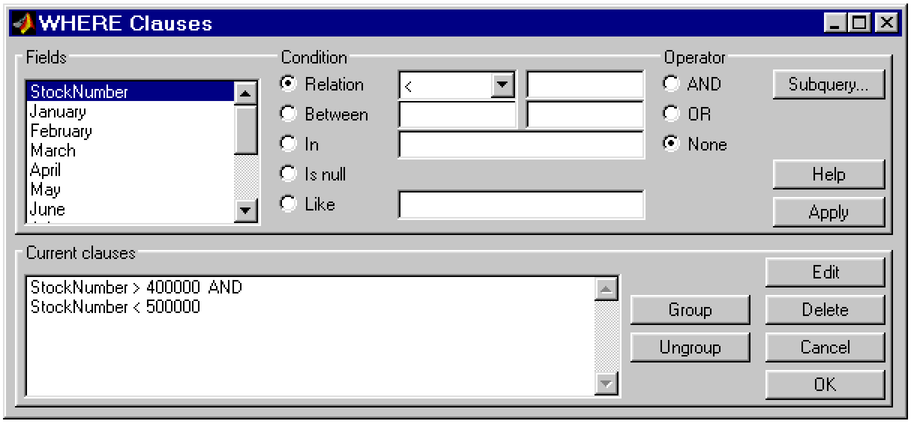
12 Save this query as `basic_where.qry`.

Grouping Statements

Use the WHERE Clauses dialog box to group query statements. In this example, modify `basic_where.qry` to retrieve data where sales in January, February, or March exceed `1500` units, if sales in each month exceed `1000` units.

To modify `basic_where.qry`:

1 Click **Where** in VQB. The WHERE Clauses dialog box appears.



2 Modify the query to retrieve data if sales in January, February, or March exceed 1500 units.

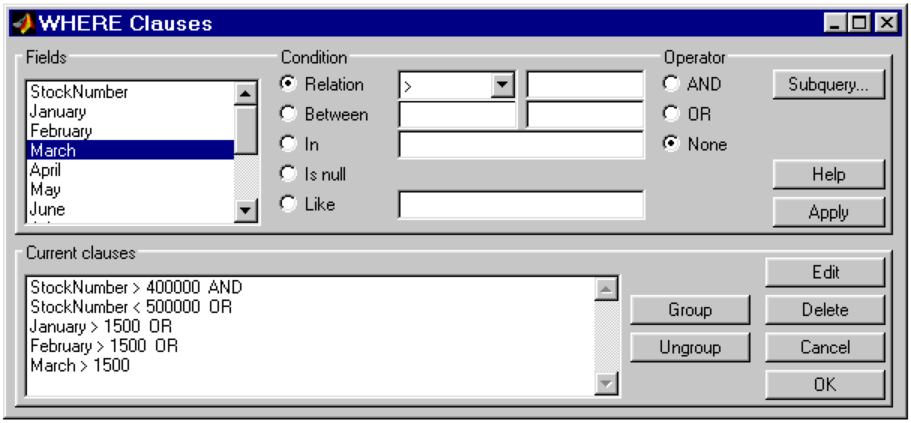
a In **Current clauses**, select `StockNumber < 500000` and click **Edit**.

b For **Operator**, select `OR` and click **Apply**.

c In **Fields**, select `January`. For **Relation**, select `>` and enter `1500` in its field. For **Operator**, select `OR`. Click **Apply**.

d Repeat step c twice, specifying `February` and `March` in **Fields**.

The WHERE Clauses dialog box now looks as follows.

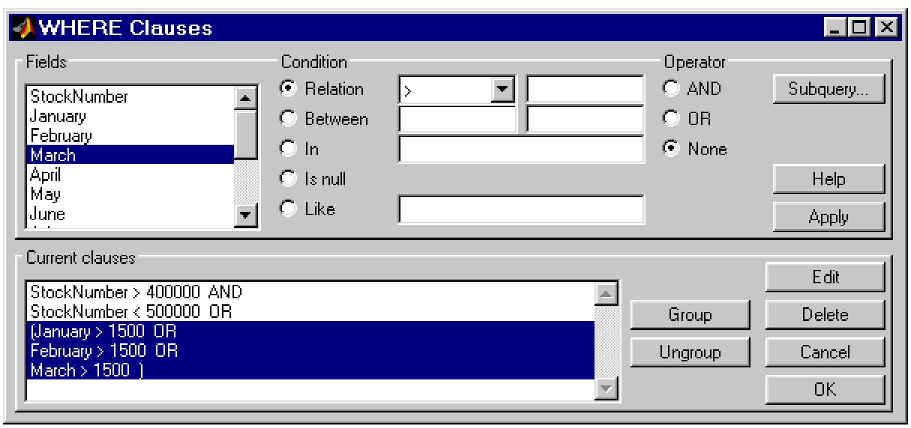


3 Group the criteria that require sales in each month to exceed 1500 units.

a In **Current clauses**, select the statement `January > 1500 OR`. Press **Shift**+click to select `February > 1500 OR` and `March > 1500` also.

b Click **Group**.

An opening parenthesis is added before `January` and a closing parenthesis is added after `March > 1500`, indicating that these statements are evaluated as a group.



4 Modify the query to retrieve data if sales in each month exceed 1000 units.

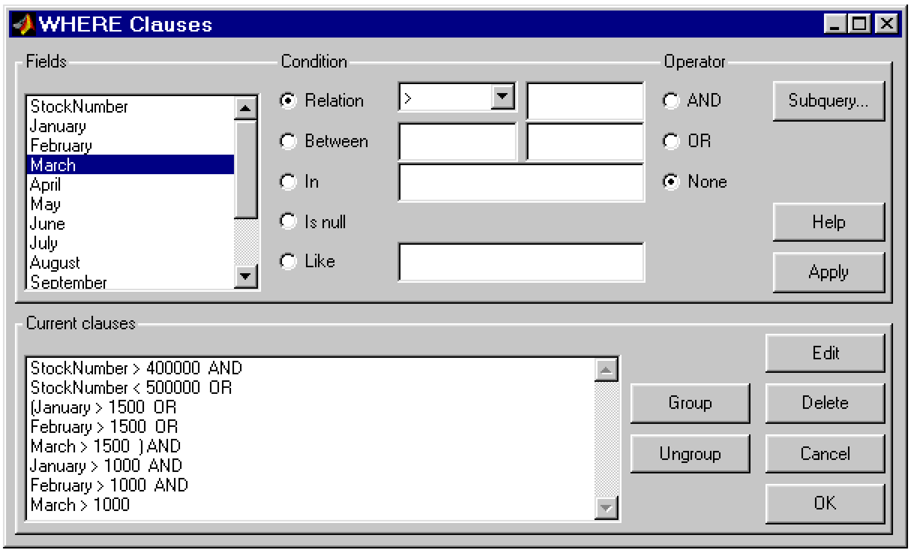
a Select `March > 1500 )` in **Current clauses** and click **Edit**.

b Select AND for **Operator** and click **Apply**.

c Select `January` in **Fields**. Select > for **Relation** and enter `1000` in its field. Select AND for **Operator**. Click **Apply**.

d Repeat step c twice, specifying `February` and `March` in **Fields**.

The WHERE Clauses dialog box now looks as follows.



e Click **OK**.

The WHERE Clauses dialog box closes. The **SQL statement** dialog box displays the modified where clause.

5 Assign the query results to the **MATLAB workspace variable** AA.

6 Click **Execute** to run the query.

7 To view the results, enter AA in the Command Window.

```
AA =

    [212569]    [2400]    [1721]    [1414]
    [400314]    [3000]    [2400]    [1800]
    [400339]    [4300]    [ NaN]    [2600]
    [400345]    [5000]    [3500]    [2800]
    [400455]    [1200]    [ 900]    [ 800]
    [400876]    [3000]    [2400]    [1500]
    [400999]    [3000]    [1500]    [1000]
```

Removing Grouping of Statements

To use the WHERE Clauses dialog box to remove grouping criteria from the previous example:

1. In **Current clauses**, select `(January > 1000 AND`.
2. Press **Shift**+click to select `February > 1000 AND` and `March > 1000)` also.
3. Click **Ungroup**.

The parentheses are removed from the statements, indicating that their grouping is removed.

Displaying Results in a Specified Order

Use **Order by** in **Advanced query options** to specify the order in which query results display.

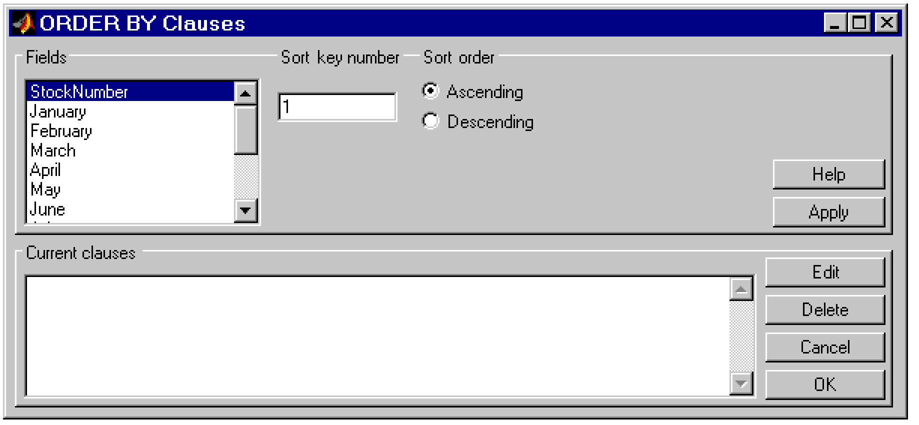
This example uses the `basic_where.qry` query you created in "Retrieving Data That Meets Specified Criteria" on page 5-36. The results of `basic_where.qry` are sorted so that January is the primary sort field, February the secondary, and March the last. Results for January and February appear in ascending order, and results for March appear in descending order.

To specify the order in which results appear in `basic_where.qry`:

1. Load `basic_where.qry`.
2. Set the **Data return format** preference to `cellarray`.
3. Set **Read NULL numbers** to `NaN`.

4 In **Advanced query options**, select **Order by**.

The ORDER BY Clauses dialog box appears.



5 Enter values for the **Sort key number** and **Sort order** fields for the appropriate **Fields**.

To specify `January` as the primary sort field and display results in ascending order:

a In **Fields**, select `January`.

b For **Sort key number**, enter `1`.

c For **Sort order**, select **Ascending**.

d Click **Apply**.

The **Current clauses** area now displays:

```
January ASC
```

6 To specify `February` as the second sort field and display results in ascending order:

a In **Fields**, select `February`.

b For **Sort key number**, enter `2`.

c For **Sort order**, select **Ascending**.

d Click **Apply**.

The **Current clauses** area now displays:

```
January ASC
February ASC
```

7 To specify `March` as the third sort field and display results in descending order:

a In **Fields**, select `March`.

b For **Sort key number**, enter `3`.

c For **Sort order**, select **Descending**.

d Click **Apply**.

The **Current clauses** area now displays:

```
January ASC
February ASC
March DESC
```

8 Click **OK**.

The ORDER BY Clauses dialog box closes. The **Order by** field and the **SQL statement** in VQB display the specified Order By clause.

9 Assign the query results to the **MATLAB workspace variable** `B`.

10 Click **Execute** to run the query.

11 To view the results, enter `B` in the Command Window. Enter `A` to display the unordered query results and compare them to `B`. Your results look as follows:

```
A =

    [400314]    [3000]    [2400]    [1800]
    [400339]    [4300]    [ NaN]    [2600]
    [400345]    [5000]    [3500]    [2800]
    [400455]    [1200]    [ 900]    [ 800]
    [400876]    [3000]    [2400]    [1500]
    [400999]    [3000]    [1500]    [1000]


B =

    [400455]    [1200]    [ 900]    [ 800]
    [400999]    [3000]    [1500]    [1000]
    [400314]    [3000]    [2400]    [1800]
    [400876]    [3000]    [2400]    [1500]
    [400339]    [4300]    [ NaN]    [2600]
    [400345]    [5000]    [3500]    [2800]
```

For `B`, results are first sorted by `January sales` in ascending order. The lowest value for `January sales`, 1200 (for item number 400455), appears first. The highest value, 5000 (for item number for 400345), appears last.

For items 400999, 400314, and 400876, `January sales` were 3000. Therefore, the second sort key `February sales`applies. February sales appear in ascending order: 1500, 2400, and 2400 respectively.

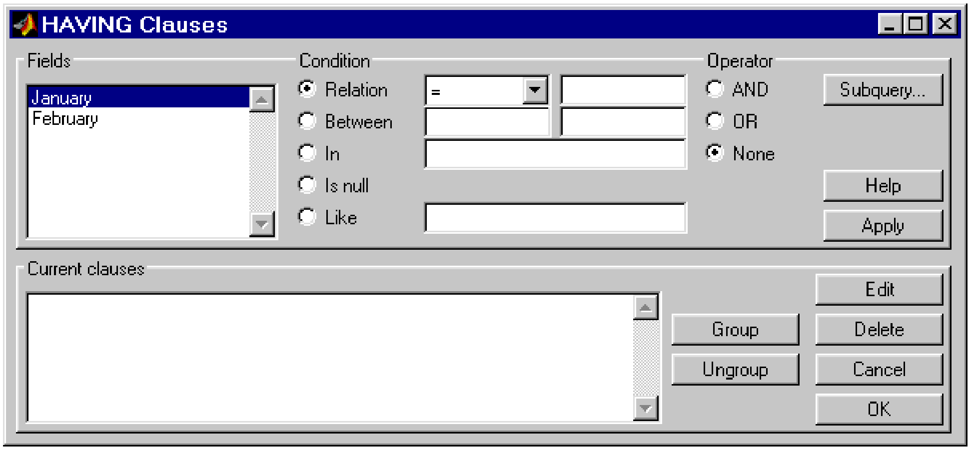
For items 400314 and 400876, `February sales` were 2400, so the third sort key, `March sales`, applies. March sales appear in descending order: 1800 and 1500, respectively.

Using Having Clauses to Refine Group by Results

Using the HAVING Clauses Dialog Box

Use the **Having** function to refine the results of a **Group By** clause.

After specifying a group-by clause in **Advanced query options**, click **Having**. The HAVING Clauses dialog box appears.



1 From the **Fields** list box, select the entry whose value to restrict.

2 Define the **Condition** for the selected field, as described in "Retrieving Data That Meets Specified Criteria" on page 5-36.

3 Select **Operator** to add another condition.

4 Click **Apply** to create the clause.

   The subquery appears in the **Current clauses** area.

5 Repeat steps 1 through 4 to add more conditions as needed.

6 Change the clauses as needed:

   - To edit a clause:

     a Select the clause from **Current clauses** and click **Edit**.

     b Modify the **Fields**, **Condition**, and **Operator** fields as needed.

     c Click **Apply**.

   - To group clauses:

     a Select the clauses to group from **Current clauses**. Press **Ctrl**+click or **Shift**+click to select multiple clauses.

     b Click **Group**. Parentheses are added around the set of clauses.

       To ungroup clauses, select the clauses and then click **Ungroup**.

- To delete a clause, Select the clause from **Current clauses** and click **Delete**. Use **Ctrl**+click or **Shift**+click to select multiple clauses.

7 Specify a subquery in the HAVING Clauses dialog box, as needed. For details, see “Creating Subqueries for Values from Multiple Tables” on page 5-49.

8 Click **OK**.

The HAVING Clauses dialog box closes. The **SQL statement** in the Visual Query Builder dialog box updates to reflect the specified having clause.

Example: Using Having Clauses

This example restricts the results from `basic_where.qry` to sales greater than 2000 for January and February:

1 In **Advanced query options**, click **Having**. The HAVING Clauses dialog box appears.

2 For `January`:

   a Select > as the **Relation Condition**.

   b Enter `2000` as the **Relation** value.

   c Select the `AND` **Operator**.

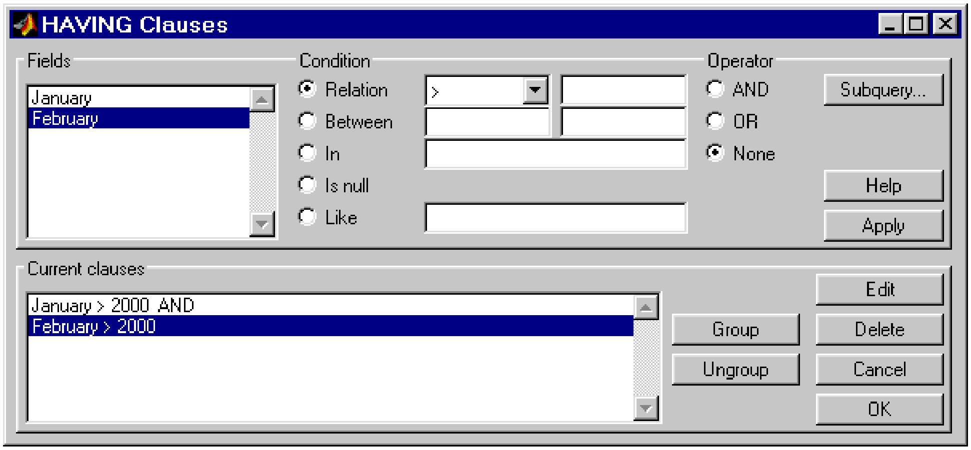
   d Click **Apply**.

3 For `February`:

   a Select > as the **Relation Condition**.

   b Enter `2000` as the **Relation** value.

   c Click **Apply**. The HAVING Clauses dialog box appears as follows.



4 Click **OK**.

The HAVING Clauses dialog box closes. The **SQL statement** field in the VQB dialog box reflects the specified Having clause.

5 Assign a MATLAB workspace variable `C`, and click **Execute** to run the query.

```
C =
    [3000]    [2400]
    [5000]    [3500]
```

Compare these results to those in "Displaying Results in a Specified Order" on page 5-43.

Creating Subqueries for Values from Multiple Tables

Use the **Where** feature in **Advanced query options** to create subqueries. Creating subqueries in this way is referred to as *nested SQL*.

This example uses `basic.qry`, which you created by selecting **Query** > **Save** and saving your query as `basic.qry` in the **File name** field.

The `salesVolume` table has sales volumes and stock number fields, but no product description field. The `productTable` has product description and stock number fields, but no sales volumes. This example retrieves the stock number for the product whose description is `Building Blocks` from the `productTable` table. It then gets the sales volume values for that stock number from the `salesVolume` table.

1 Load `basic.qry`.

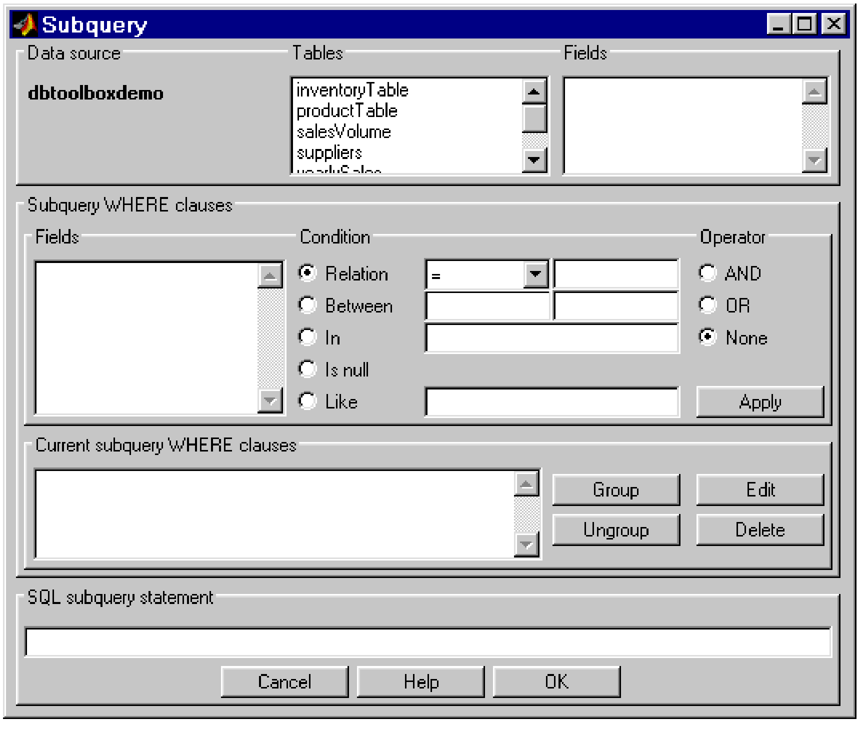
2 Set the **Data return format** Preference to `cellarray` and **Read NULL numbers as** to `NaN`.

3 Click **Where** in **Advanced query options**.

The WHERE Clauses dialog box appears.

4 Click **Subquery**.

The Subquery dialog box appears.



5 In **Tables**, select `productTable`, which includes the association between the stock number and the product description. The fields in that table appear.

6 In **Fields**, select `stockNumber`, the field that is common to this table and the table from which you are retrieving results.

The statement `SELECT stockNumber FROM productTable` is created in the **SQL subquery statement**.

7 Limit the query to product descriptions that are Building Blocks.

a In **Fields** in **Subquery WHERE clauses**, select `productDescription`.

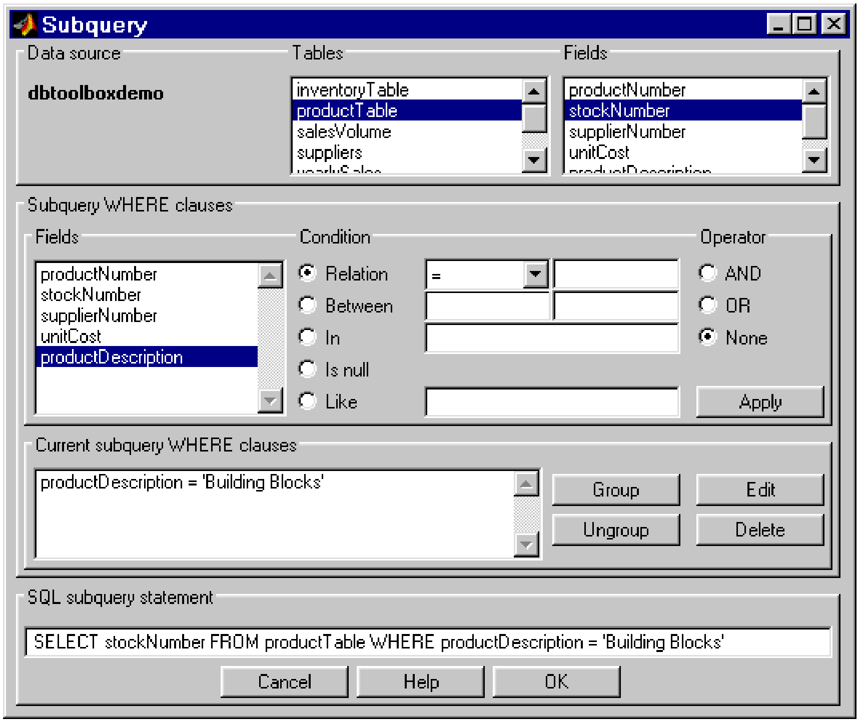
b For **Condition**, select **Relation**.

c In the drop-down list to the right of **Relation**, select `=`.

d In the field to the right of the drop-down list, enter `'Building Blocks'`.

e Click **Apply**.

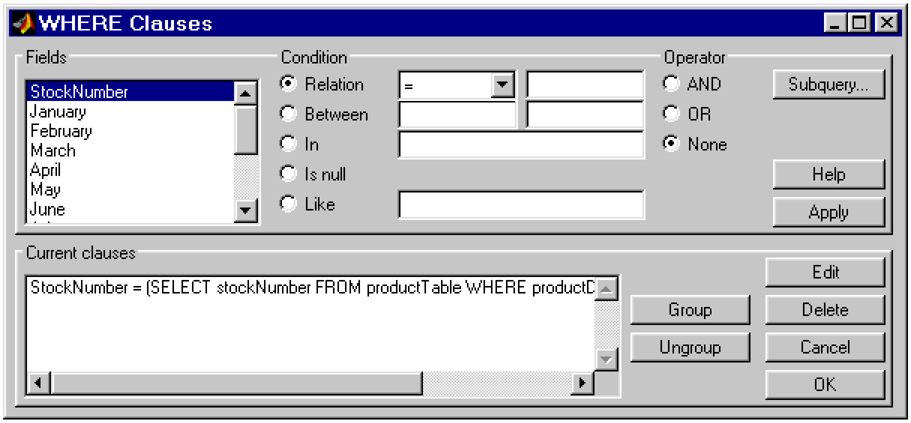
The clause appears in the **Current subquery WHERE clauses** field and is added to the **SQL subquery statement**.



8 Click **OK** to close the Subquery dialog box.

9 In the WHERE Clauses dialog box, click **Apply**.

This updates the **Current clauses** area using the subquery criteria specified in steps 3 through 8.



10 In the WHERE Clauses dialog box, click **OK**.

The WHERE Clauses dialog box closes. The **SQL statement** in the VQB dialog box updates.

11 Assign the query results to the **MATLAB workspace variable** `C`.

12 Click **Execute**.

13 Type `C` at the prompt in the Command Window to see the results.

```
C =

    [400345]    [5000]    [3500]    [2800]
```

14 The results are for item 400345, which has the product description `Building Blocks`, although that is not evident from the results. Create and run a query to verify that the product description is `Building Blocks`:

a For **Data source**, select `dbtoolboxdemo`.

b In **Tables**, select `productTable`.

c In **Fields**, select `stockNumber` and `productDescription`.

d Assign the query results to the **MATLAB workspace variable** `P`.

e Click **Execute**.

f Type `P` at the prompt in the Command Window to view the results.

```
P =

    [125970]    'Victorian Doll'
    [212569]    'Train Set'
    [389123]    'Engine Kit'
    [400314]    'Painting Set'
    [400339]    'Space Cruiser'
    [400345]    'Building Blocks'
    [400455]    'Tin Soldier'
    [400876]    'Sail Boat'
    [400999]    'Slinky'
    [888652]    'Teddy Bear'
```

The results show that item 400345 has the product description `Building Blocks`. In the next section, you create a query that includes `product description` in the results.

Note: You can include only one subquery in a query using VQB; you can include multiple subqueries using Database Toolbox functions.

Creating Queries That Include Results from Multiple Tables

A query whose results include values from multiple tables is said to perform a *join* operation in SQL.

This example retrieves sales volumes by product description. It is like the one in “Creating Subqueries for Values from Multiple Tables” on page 5-49, but this example creates a query that returns `product description` rather than `stock number`.

The `salesVolume` table has `sales volume` and `stock number` fields, but no `product description` field. The `productTable` table has `product description` and `stock number` fields, but no `sales volume` field. To create a query that retrieves data from both tables and equates the stock number from `productTable` with the stock number from `salesVolume`:

1 Set the **Data return format** preference to `cellarray` and the **Read NULL numbers as** preference to `NaN`.

2 For **Data operation**, click **Select**.

3 For **Data source**, select `dbtoolboxdemo`.

   The **Catalog**, **Schema**, and **Tables** for `dbtoolboxdemo` appear.

   Do not specify **Catalog** or **Schema**.

4 In **Tables**, select the tables from which you want to retrieve data. For this example, press **Ctrl**+click and select both `productTable` and `salesVolume`.

   The fields (columns) in those tables appear in **Fields**. Field names appear in the format `tableName.fieldName`. Therefore, `productTable.stockNumber` indicates the stock number in the product table and `salesVolume.StockNumber` indicates the stock number in the sales volume table.

5 In **Fields**, press **Ctrl**+click to select the following fields:

   - `productTable.productDescription`
   - `salesVolume.January`
   - `salesVolume.February`
   - `salesVolume.March`

6 In this example, the Where clause equates the `productTable.stockNumber` with the `salesVolume.StockNumber`, so that product description is associated with sales volumes in the query results.

   In **Advanced query options**, click **Where** to associate fields from different tables. The WHERE Clauses dialog box appears.

7 In the WHERE clauses dialog box:

   a In **Fields**, select `productTable.stockNumber`.

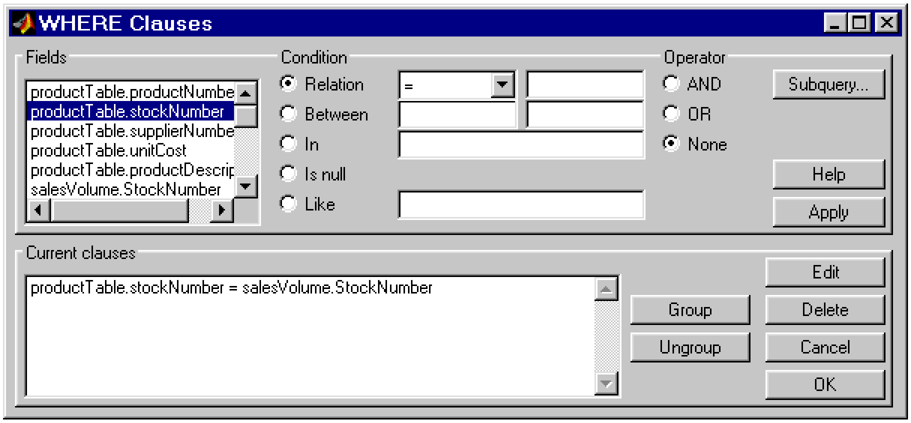
   b For **Condition**, select **Relation**.

   c In the drop-down list to the right of **Relation**, select `=`.

   d In the field to the right of the drop-down list, enter `salesVolume.StockNumber`.

   e Click **Apply**.

      The clause appears in the **Current clauses** field.



f Click **OK** to close the WHERE Clauses dialog box. The **Where** field and **SQL statement** in VQB display the Where clause.

8 Assign the query results to the **MATLAB workspace variable** P1.

9 Click **Execute** to run the query.

10 Type P1 in the Command Window.

```
P1 =

    'Victorian Doll'      [1400]     [1100]     [ 981]
    'Train Set'           [2400]     [1721]     [1414]
    'Engine Kit'          [1800]     [1200]     [ 890]
    'Painting Set'        [3000]     [2400]     [1800]
    'Space Cruiser'       [4300]     [ NaN]     [2600]
    'Building Blocks'     [5000]     [3500]     [2800]
    'Tin Soldier'         [1200]     [ 900]     [ 800]
    'Sail Boat'           [3000]     [2400]     [1500]
    'Slinky'              [3000]     [1500]     [1000]
    'Teddy Bear'          [ NaN]     [ 900]     [ 821]
```

Additional Advanced Query Options

For details about advanced query options, choose an option and click **Help** in its dialog box. For example, click **Group by** in **Advanced query options**, and then click **Help** in the Group by Clauses dialog box.

Retrieving BINARY and OTHER Data Types

This example shows how to retrieve data of types `BINARY` and `OTHER`, which may require manipulation before it can undergo MATLAB processing. To retrieve images using the `dbtoolboxdemo` data source and a sample file that parses image data, *`matlabroot`*`/toolbox/database/vqb/parsebinary.m`:

1. For **Data Operation**, select **Select**.
2. In **Data source**, select `dbtoolboxdemo`.
3. In **Tables**, select `Invoive`.
4. In **Fields**, select `InvoiceNumber` and `Receipt` (which contains bitmap images).
5. Select **Query** > **Preferences**.
6. In the **Data return format** field, specify `cellarray`.
7. As the **MATLAB workspace variable**, specify `A`.
8. Click **Execute** to run the query.
9. Type `A` in the Command Window to view the query results.

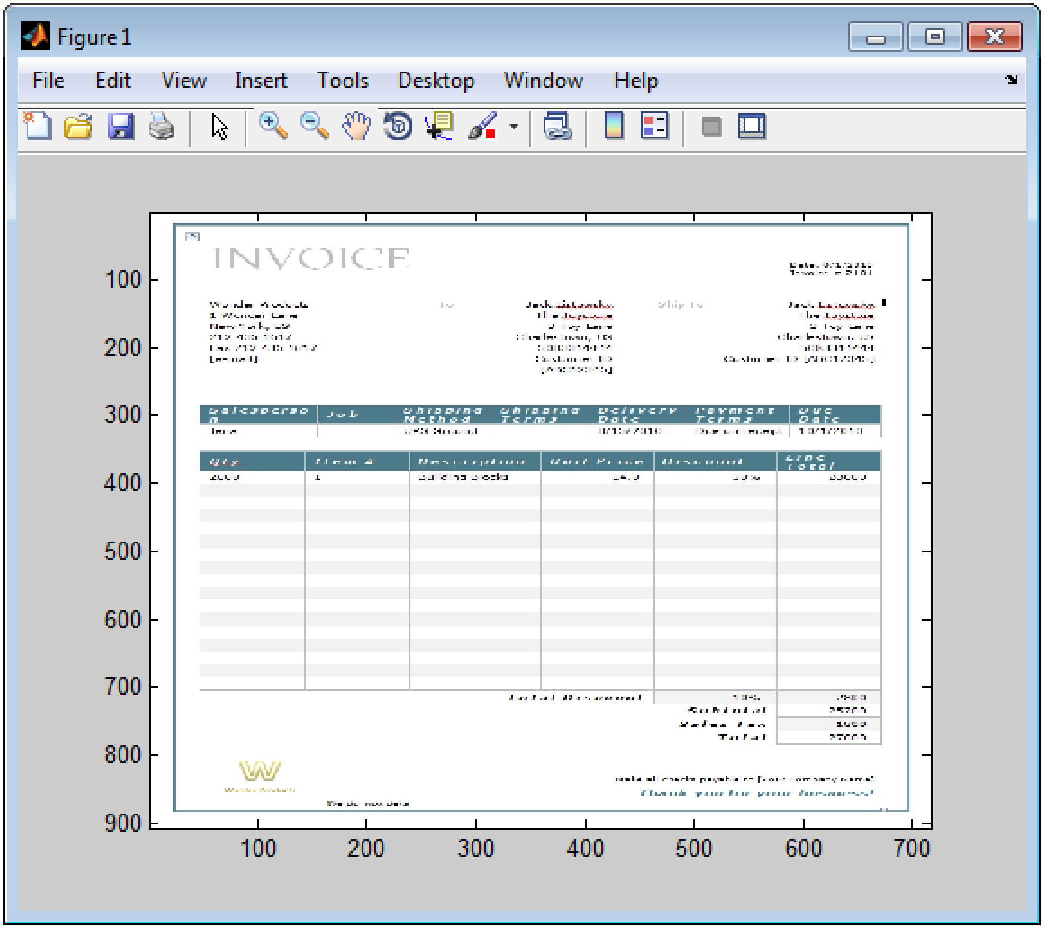
```
A =

    [1]    [21626x1 int8]
    [2]    [21626x1 int8]
    [3]    [21722x1 int8]
    [4]    [21626x1 int8]
    [5]    [21626x1 int8]
    [6]    [21626x1 int8]
    [7]    [21626x1 int8]
    [8]    [21626x1 int8]
    [9]    [21626x1 int8]
```

10. Assign the first element in A to the variable `photo`.

```
photo = A{1,2};
```

11. Make sure your current folder is writable.
12. Run the sample program `parsebinary`, which writes the retrieved data to a file, strips ODBC header information, and displays `photo` as a bitmap image.

```
cd I:\MATLABFiles\myfiles
parsebinary(photo, 'BMP');
```



For details about `parsebinary`, enter `help parsebinary`, or view the `parsebinary` file in the MATLAB Editor/Debugger by entering `open parsebinary` in the Command Window.

Importing and Exporting Boolean Data

In this section...

Import Boolean Data from Databases

`BOOLEAN` data is imported from databases into the MATLAB workspace as data type `logical`. This data has a value of `0` (false) or `1` (true), and is stored in a cell array or structure.

This example imports data from the `Invoice` table in the `dbtoolboxdemo` database into the MATLAB workspace.

1. Set **Data return format** to `cellarray`.
2. For **Data operation**, choose **Select**.
3. In **Data source**, select `dbtoolboxdemo`.
4. In **Tables**, select `Invoice`.
5. In **Fields**, select `Paid` and `InvoiceNumber`.
6. Assign the query results to the **MATLAB workspace variable** `D`.
7. Click **Execute** to run the query.

   VQB retrieves a 10-by-2 array.
8. Enter `D` in the Command Window. 10 records are returned:

```
D =

    [ 2101]    [0]
    [ 3546]    [1]
    [33116]    [1]
    [34155]    [0]
    [34267]    [1]
    [37197]    [1]
    [37281]    [0]
    [41011]    [1]
```

```
[61178]      [0]
[62145]      [1]
```

9 Compare these results to the data in Microsoft Access.

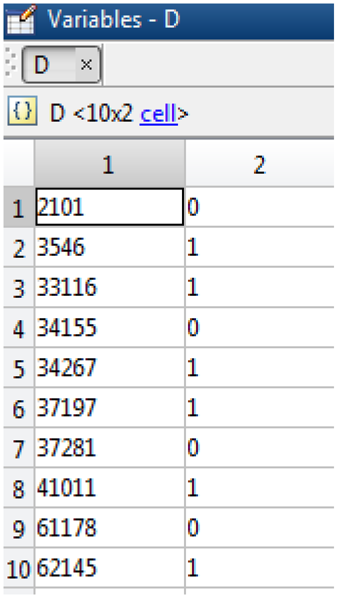
invoice

InvoiceNum	InvoiceDate	ProductNumber	Paid	Receipt
2101	8/1/2010	1	☐	Bitmap Image
3546	3/1/2010	2	☑	Bitmap Image
33116	5/15/2011	3	☑	Bitmap Image
34155	7/12/2011	4	☐	Bitmap Image
34267	7/22/2011	5	☑	Bitmap Image
37197	9/3/2011	6	☑	Bitmap Image
37281	9/21/2011	7	☐	Bitmap Image
41011	12/12/2011	8	☑	Bitmap Image
61178	1/15/2012	9	☐	Bitmap Image
62145	1/23/2012	10	☑	Bitmap Image

invoice

Field Name	Data Type
InvoiceNumber	Number
InvoiceDate	Date/Time
ProductNumber	Number
Paid	Yes/No
Receipt	OLE Object

10 In the VQB **Data** area, double-click `D` to view its contents in the Variables editor.


Variables - D
D
D <10x2 cell>
1
2
1 2101 0
2 3546 1
3 33116 1
4 34155 0
5 34267 1
6 37197 1
7 37281 0
8 41011 1
9 61178 0
10 62145 1

Exporting Boolean Data to Databases

Logical data is exported from the MATLAB workspace to a database as type `BOOLEAN`. This example adds two rows of data to the `Invoice` table in the `dbtoolboxdemo` database.

1 In the MATLAB workspace, create `I`, the structure you want to export.

```
I.InvoiceNumber{1,1}=456789;
I.Paid{1,1}=logical(0);
I.InvoiceNumber{2,1}=987654;
I.Paid{2,1}=logical(1);
```

2 For **Data operation**, choose **Insert**.

3 In **Data source**, select `dbtoolboxdemo`.

4 In **Tables**, select `Invoice`.

5 In **Fields**, select `Paid` and `InvoiceNumber`.

6 Assign results to the **MATLAB workspace variable** `I`.

7 Click **Execute** to run the query.

   VQB inserts two new rows into the `Invoice` table.

View the table in Microsoft Access to verify that the data was correctly inserted.

invoice

InvoiceNum	InvoiceDate	ProductNumber	Paid	Receipt
987654			☑	
456789			☐	
2101	8/1/2010	1	☐	Bitmap Image
3546	3/1/2010	2	☑	Bitmap Image
33116	5/15/2011	3	☑	Bitmap Image
34155	7/12/2011	4	☐	Bitmap Image
34267	7/22/2011	5	☑	Bitmap Image
37197	9/3/2011	6	☑	Bitmap Image
37281	9/21/2011	7	☐	Bitmap Image
41011	12/12/2011	8	☑	Bitmap Image
61178	1/15/2012	9	☐	Bitmap Image
62145	1/23/2012	10	☑	Bitmap Image

Saving Queries in Files

In this section...

About Generated Files

Select **Query** > **Generate MATLAB File** to create a file that contains the equivalent Database Toolbox functions required to run an existing query that was created in VQB. Edit the file to include MATLAB or related toolbox functions, as needed. To run the query, execute the file.

The following is an example of a file generated by VQB:

```
% Set preferences with setdbprefs.
s.DataReturnFormat = 'cellarray';
s.ErrorHandling = 'store';
s.NullNumberRead = 'NaN';
s.NullNumberWrite = 'NaN';
s.NullStringRead = 'null';
s.NullStringWrite = 'null';
s.JDBCDataSourceFile = '';
s.UseRegistryForSources = 'yes';
s.TempDirForRegistryOutput = '';
s.FetchInBatches = 'yes';
s.FetchBatchSize = '10000'
setdbprefs(s)

% Make connection to database. Note that the password has been omitted.
% Using ODBC driver.
conn = database('dbtoolboxdemo','','password');

% Read data from database.
e = exec(conn,'SELECT ALL StockNumber,January,February FROM salesVolume');
e = fetch(e);
close(e)


 Close database connection.
close(conn)
```

VQB Query Elements in Generated Files

The following VQB query elements do not appear in generated files:

- Generated code files do not include MATLAB workspace variables to which you assigned query results in the VQB query. The file assigns the query results to `e`; access these results using the variable `e.Data`. For example, you can add a statement to the file that assigns a variable name to `e.Data` as follows:

  ```
  myVar = e.Data
  ```

- For security reasons, generated files do not include passwords required to connect to databases. Instead, the `database` statement includes `'password'` as a placeholder. To run files to connect to databases that require passwords, substitute your password for `password` in the `database` statement.

Saving Queries

1. Click **Query** > **Save**. The Save SQL Statement dialog box appears.
2. Enter a name (without spaces) for the query into the **File name** field and click **Save**. Save the query as `basic.qry`.

Note: When you save a **Select** query (a query that imports data), MATLAB does not save your specified preferences or the workspace variable that contains the query results. This prevents you from inadvertently overwriting an existing variable in the MATLAB workspace when you run a saved query.

When you save an **Insert** query (a query that exports data), MATLAB saves the workspace variable whose data you exported, but does not save your preferences.

Running Saved Queries

1. Click **Query** > **Load**. The Load SQL Statement dialog box appears.
2. Select the name of the query you want to load and click **Open**. The VQB fields reflect the values for the saved query.
3. Run a **Select** query to import data into the MATLAB workspace, or an **Insert** query to export data from the MATLAB workspace.

- To run a **Select** query, use the **MATLAB workspace variable** field to assign a variable to the data and click **Execute**.
- For an **Insert** query, the saved query may include a workspace variable, which appears as part of the **MATLAB command** field. Type that variable name or enter a new name in the **MATLAB workspace variable** field. Press **Return** or **Enter** to see the updated **MATLAB command**.

4 Click **Execute** to run the query.

Tip You can generate a file that runs the query from the Command Window in the future. For details, see "Saving Queries in Files" on page 5-62 in the Database Toolbox documentation.

Editing Queries

Edit a query using one of the following options:

- Changing your selections.
- Editing the **SQL statement** field.
- Editing the **MATLAB command** field.

6

Using Database Toolbox Functions

Getting Started with Database Toolbox Functions

The following sections provide examples of how to use Database Toolbox functions. MATLAB files that include functions used in some of these examples are available in `matlab/toolbox/database/dbdemos`.

Follow these simple examples consecutively when you first start using the product. Once you are familiar with Database Toolbox usage, refer to these examples as needed.

Import Data from Databases into MATLAB

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

Connect to the Database

Connect to the Microsoft Access database with the data source name `dbtoolboxdemo` using native ODBC.

```
conn = database.ODBCConnection('dbtoolboxdemo','','');
```

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

Import Data Using a 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 object `conn`.

```
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 the SQL Query

Connect to the Microsoft Access database with the data source name `dbtoolboxdemo` using the native ODBC interface.

```
conn = database.ODBCConnection('dbtoolboxdemo','','');
```

Create an SQL script file named `salesvolume.sql` with this SQL query. This SQL query uses multiple joins to join these tables in the `dbtoolboxdemo` 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 =

  cursor with properties:

        Attributes: []
              Data: {6x13 cell}
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: 'SELECT  salesvolume.January ,   salesvolume.February ,   salesvolume.March ,   salesv...'
           Message: ''
              Type: 'Database Cursor Object'
         ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
            Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
         Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
             Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

`results` contains a cursor 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 the Database Connection

Close the cursor 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 6-8

More About

- “Connecting to a Database Using the Native ODBC Interface” on page 3-18

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 a Query Using a 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 the native ODBC interface. 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.ODBCConnection('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 the native ODBC interface. For example, the following code assumes that you are connecting to a data source named `dbtoolboxdemo` with a blank user name and password.

```
conn = database.ODBCConnection('dbtoolboxdemo','','');
```

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 a Query Using a 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 a 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 6-4

More About

- “Connecting to a Database Using the Native ODBC Interface” on page 3-18

Roll Back and Commit Data in a 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 6-22
- "Replace Existing Data in a Database" on page 6-25

Change the 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 6-39

Create a Table and Add a 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 for the database connection object `conn`. The cursor object `curs` 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 6-39

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 the Database

Create the database connection `conn` to a Microsoft Access database using the native ODBC interface and the data source name `dbtoolboxdemo`. This database contains the table `inventoryTable` with the column `productNumber`.

```
conn = database.ODBCConnection('dbtoolboxdemo','','');
```

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 a 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 a Record Using a 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.

```
curs = exec(conn,'select * from inventoryTable');
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 the 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 6-4

More About

- “Connecting to a Database Using the Native ODBC Interface” on page 3-18

Roll Back Data After Updating a 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);
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

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 6-22
- "Replace Existing Data in a Database" on page 6-25

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`.

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 `dbtoolboxdemo` with blank user name and password.

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

   Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.
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.

yearlySales

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 the MATLAB Workspace" on page 6-27
- "Export Data Using Bulk Insert" on page 6-31
- "Replace Existing Data in a Database" on page 6-25

More About

- "Inserting Data Using the Command Line" on page 2-197

Replace Existing Data in a Database

This example shows how to update a value of the `Month` column in the table `yearlySales` using the data source named `dbtoolboxdemo`. To access the example where you import the values of the `Month` column, see “Export Data to New Record in Database” on page 6-22.

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 JDBC/ODBC bridge. Here, this code assumes that you are connecting to a data source named `dbtoolboxdemo` with blank user name and password.

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

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.

yearlySales

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 6-22
- "Export Multiple Records from the MATLAB Workspace" on page 6-27

Export Multiple Records from the MATLAB Workspace

This example does the following:

1 Imports monthly sales figures for all products from the `dbtoolboxdemo` 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`.

1 Create a database connection `conn` to the Microsoft Access database using the JDBC/ODBC bridge. Here, this code assumes that you are connecting to a data source named `dbtoolboxdemo` with blank user name and password.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

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
```

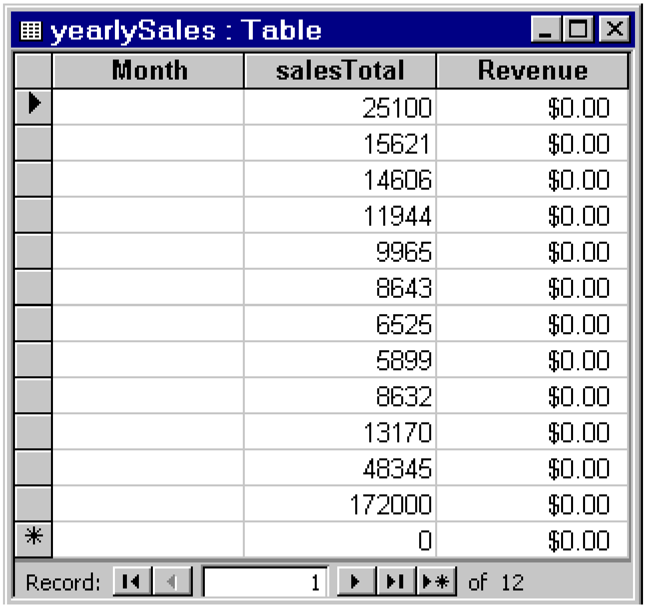
10. Create a cell array colnames containing the column name for inserting the data.

```
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.



yearlySales : Table

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 6-22

More About

- “Inserting Data Using the Command Line” on page 2-197

Export Data Using Bulk Insert

About Bulk Insert Functionality

Many ways exist to insert data into your database using the command line. You can use `datainsert`, `fastinsert`, or `insert`. For best performance with large volumes of data, use `datainsert` or `fastinsert`.

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. Additionally, you can insert data with special characters such as double quotes with this file. Bulk insert provides performance gains by using the bulk insert utilities that are native to different database systems. For details about working with large data sets, see "Working with Large Data Sets" on page 2-199.

Bulk Insert into Oracle

This example shows how to export data to the Oracle server using bulk insert. For this example, use a data file on the local machine where Oracle is installed.

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`.

```
e = exec(conn,['create table BULKTEST (salary number, '...
'player varchar2(25), signed varchar2(25), '...
'team varchar2(25))']);
close(e)
```

3 Enter data records. A sample record appears as follows.

```
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.

Tip When connecting to a database on a remote machine, you must write this file to the remote machine. Oracle has problems trying to read 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.

```
e = exec(conn,...
   'create or replace directory ext as ''C:\\Temp''');
close(e)
```

7 Delete the temporary table if it exists.

```
e = exec(conn,'drop table testinsert');
try,close(e),end
```

8 Create a temporary table and bulk insert it into the table BULKTEST.

```
e = 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(e)
e = exec(conn,'insert into BULKTEST select * from testinsert');
close(e)
```

9 Confirm the number of rows and columns in BULKTEST.

```
e = exec(conn, 'select * from BULKTEST');
results = fetch(e)

results =

  cursor with properties:

        Attributes: []
              Data: {10000x4 cell}
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: 'select * from BULKTEST'
```

```
          Message: []
             Type: 'Database Cursor Object'
        ResultSet: [1x1 oracle.jdbc.driver.OracleResultSetImpl]
           Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
        Statement: [1x1 oracle.jdbc.driver.OracleStatementWrapper]
            Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]

columnnames(results)

ans =

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

10 Close the connection.

```
close(conn)
```

Bulk Insert into Microsoft SQL Server 2005

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.

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

3 Enter data records. A sample record appears as follows.

```
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.

Tip When connecting to a database on a remote machine, you must write this file to the remote machine. Microsoft SQL Server has problems trying to read 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.

```
e = exec(conn,['bulk insert BULKTEST from '...
'''c:\temp\tmp.txt''with (fieldterminator = ''\t'', '...
'rowterminator = ''\n'')']);
```

7 Confirm the number of rows and columns in BULKTEST.

```
e = exec(conn, 'select * from BULKTEST');
results = fetch(e)

results =

  cursor with properties:

        Attributes: []
              Data: {10000x4 cell}
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: 'select * from BULKTEST'
           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]

columnnames(results)

ans =

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

8 Close the connection.

```
close(conn)
```

Bulk Insert into MySQL

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.

```
e = exec(conn,['create table BULKTEST (salary decimal, '...
   'player varchar(25), signed_date varchar(25), '...
   'team varchar(25))']);
close(e)
```

3 Create a data record, such as the one that follows.

```
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.

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. Note the use of `local infile`.

```
e = exec(conn,['load data local infile '...
   ' ''C:\\temp\\tmp.txt'' into table BULKTEST '...
   'fields terminated by ''\t'' lines terminated '...
   'by ''\n''']);
close(e)
```

7 Confirm the number of rows and columns in BULKTEST.

```
e = exec(conn, 'select * from BULKTEST');
```

```
results = fetch(e)

results =

  cursor with properties:

         Attributes: []
               Data: {10000x4 cell}
     DatabaseObject: [1x1 database]
           RowLimit: 0
           SQLQuery: 'select * from BULKTEST'
            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: [1x1 com.mathworks.toolbox.database.fetchTheData]

columnnames(results)

ans =

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

8 Close the connection.

```
close(conn)
```

See Also

close | database | exec | fetch

Related Examples

- "Export Data to New Record in Database" on page 6-22

More About

- "Inserting Data Using the Command Line" on page 2-197

Retrieve Image Data Types

This example retrieves images from the `dbtoolboxdemo` data source using a sample file that parses image data, *`matlabroot`*`/toolbox/database/vqb/parsebinary.m`.

1 Connect to the `dbtoolboxdemo` data source.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

2 Specify `cellarray` as the data return format preference.

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

3 Import the `InvoiceNumber` and `Receipt` columns of data from the `invoice` table.

```
curs = exec(conn,'select InvoiceNumber, Receipt from Invoice')
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]                  []
```

Note: Some `OTHER` data type fields may be empty, indicating that the data could not pass through the JDBC/ODBC bridge.

5 Assign the image element you want to the variable `receipt`.

```
receipt = curs.Data{1,2};
```

6 Run `parsebinary`. This program writes the 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 output of `parsebinary` can be written to it.

```
cd 'I:\MATLABFiles\myfiles'
parsebinary(receipt,'BMP');
```

For details about `parsebinary`, enter `help parsebinary` or view its file in the MATLAB Editor/Debugger by entering `open parsebinary`.

See Also

`database` | `exec` | `fetch` | `setdbprefs`

Related Examples

- "Import Data from Databases into MATLAB" on page 6-4

Display Database Metadata

This example shows how to display database information for database connection objects using the command line. To view the database structure quickly, use Database Explorer to explore the tables and column names. Here, metadata refers to the information about the database structure and various database properties.

Create the Database Connection

Create a database connection `conn` using the `dbtoolboxdemo` data source.

```
conn = database('dbtoolboxdemo','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 the Database Metadata Object

Create a database metadata object `dbmeta` using `conn`.

```
dbmeta = dmd(conn)
```

```
dbmeta =

  DMDHandle: [1x1 sun.jdbc.odbc.JdbcOdbcDatabaseMetaData]
```

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`.

Display the properties `props` this database supports using `dbmeta`.

```
props = supports(dbmeta)
```

```
props =

        AlterTableWithAddColumn: 1
       AlterTableWithDropColumn: 1
            ANSI92EntryLevelSQL: 1
...
```

A `1` for a given property indicates that the database supports that property. `0` means that the database does not support the property.

For details about properties that the database supports, 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 the Database Connection

```
close(conn)
```

See Also

`dmd` | `get` | `resultset` | `rsmd` | `supports` | `tables`

Related Examples

- "Display Information About Imported Data" on page 6-56

More About

- "Working with Database Explorer" on page 4-2

About Database Toolbox Objects and Methods

This toolbox is an object-oriented application. You do not need to be familiar with the product's object-oriented implementation to use it; this information is provided for reference purposes.

Database Toolbox software includes the following objects:

- Cursor
- Database
- Database metadata
- Resultset
- Resultset metadata

Each object has its own method folder, whose name begins with an `@` sign, in the *`matlabroot`*`/toolbox/database/database` folder. Functions in the folder for each object provide methods for operating on the object.

Object-oriented characteristics of the toolbox enable you to:

- Use constructor functions to create and return information about objects.

  For example, to create a cursor object containing query results, run the `fetch` function. The object and stored information about the object are returned. Because objects are MATLAB structures, you can view elements of the returned object.

This example uses the `fetch` function to create a cursor object `curs`.

```
curs = exec(conn, 'select productdescription from producttable');

curs = fetch(curs)

curs =
      Attributes: []
            Data: {10x1 cell}
  DatabaseObject: [1x1 database]
        RowLimit: 0
        SQLQuery: 'select productdescription from producttable'
         Message: []
            Type: 'Database Cursor Object'
       ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
          Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
       Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
           Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

View the contents of the `Data` element 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'
```

- Use overloaded functions.

  Objects allow the use of overloaded functions, which simplify usage because you only need to use one function to operate on objects.For example, use the `get` function to view properties of an object.

- Create custom methods that operate on Database Toolbox objects and store them in the MATLAB workspace.

Call a 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 stored procedure `getSupplierInfo` is defined in the Microsoft SQL Server database. 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 suppliers where city = @cityName
END
GO
```

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 the Database Connection

Using the JDBC interface, connect to the Microsoft SQL Server database called `'test_db'` with the user name `'root'` and password `'matlab'` using port number `1234`. This example assumes your database server is located on the machine `servername`.

```
conn = database('test_db','root','matlab',...
      'Vendor','Microsoft SQL Server',...
      'Server','servername','PortNumber',1234)
```

```
conn =

  connection with properties:
```

```
       Instance: 'test_db'
       UserName: 'root'
         Driver: []
            URL: []
    Constructor: [1x1 com.mathworks.toolbox.database.databaseConnect]
        Message: []
         Handle: [1x1 com.microsoft.sqlserver.jdbc.SQLServerConnection]
        TimeOut: 0
     AutoCommit: 'on'
           Type: 'Database Object'
```

`database` returns `conn`, a connection `Database Object` for the `'test_db'` database.

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Call the Stored Procedure

To return the result set in table format, use `setdbprefs` to set `'DataReturnFormat'` to `'table'`.

```
setdbprefs('DataReturnFormat','table')
```

Call the stored procedure, `getSupplierInfo`, to return supplier information for the city of `New York` using `exec` with `conn`.

```
sqlquery = '{call getSupplierInfo(''New York'')}';
curs = exec(conn,sqlquery)
```

```
curs =

  cursor with properties:

         Attributes: []
               Data: 0
     DatabaseObject: [1x1 database]
           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 `Database Cursor Object`, `curs`, containing the supplier information.

Retrieve Output Data from the Stored Procedure

Retrieve supplier data from `curs` using `fetch`.

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

```
curs =

  cursor with properties:

        Attributes: []
              Data: [3x5 table]
    DatabaseObject: [1x1 database]
          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 from calling the stored procedure, `getSupplierInfo`, in table format.

Display the supplier data in table format by accessing the contents of the `Data` element of `curs`.

```
curs.Data
```

```
ans =

    SupplierNumber       SupplierName         City
    --------------    ------------------    ----------
    1001              'Wonder Products'     'New York'
    1006              'ACME Toy Company'    'New York'
    1012              'Aunt Jemimas'        'New York'


        Country          FaxNumber
    ---------------    --------------
    'United States'    '212 435 1617'
    'United States'    '212 435 1618'
```

```
    'USA'                 '14678923104'
```

Close the 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

Run a 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 the Database Connection

Connect to Microsoft SQL Server. 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.ODBCConnection('MS SQL Server','username','pwd');
```

Execute the 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 the 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`

Managing Memory to Import Data

For basic data import, you can use the Database Explorer app. For more complex queries, managing memory issues, and working with large data sets, use the functions `exec` and `fetch` to import data into the MATLAB workspace.

Importing Data Is a Two-Step Process

Importing data is a two-step process consisting of executing a query and retrieving the results into the MATLAB workspace.

Running `exec` executes the SQL query. If you are using the native ODBC interface, `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, `exec` moves the results into the Java heap.

Then, running `fetch` moves the results from RAM or the Java heap to the MATLAB workspace. `fetch` controls the number of rows imported into MATLAB memory using the database preferences `'FetchInBatches'` and `'FetchBatchSize'`.

When you import large data sets, there are two main ways to manage the memory: limiting the number of rows returned by the executed query and importing data in batches.

Managing Memory by Limiting the Number of Rows

To manage memory by limiting the number of rows that are imported from an executed query, use one of these options.

To manage RAM or Java heap memory depending on the driver:

- You can use the name-value pair argument `'maxRows'` in `exec`.
- You can modify the query by adding LIMIT or a similar SQL syntax at the end of the SQL statement. This SQL statement assumes a MySQL database connection.

  ```
  sqlquery = SELECT * FROM productTable LIMIT 5;
  ```

To manage MATLAB memory:

- You can use the `rowlimit` input argument in `fetch`.
- If you are using a SQL script, you can use the `RowInc` input argument in `runsqlscript`.

- If you are using the MATLAB interface to SQLite, you can use the `rowlimit` input argument in `fetch`.

Managing MATLAB Memory by Importing Data in Batches

To import data in batches, use the database preferences `'FetchInBatches'` and `'FetchBatchSize'` or the row limit. These methods ensure that MATLAB memory is not exceeded when importing data from the Java heap into the MATLAB workspace.

When you enable `'FetchInBatches'`, `fetch` imports all rows into MATLAB memory in batches. `'FetchBatchSize'` specifies the number of rows in a batch.

When you use the `rowlimit` input argument in `fetch`, the software ignores the `'FetchInBatches'` database preference and imports the first number of rows into the MATLAB workspace. This example imports only the first 100 rows and points the cursor to row 101 in the resultset.

```
fetch(curs,100)
```

The next run of `fetch` imports the next 100 rows and advances the cursor to 201, and so on.

```
 fetch(curs,100)
```

Given that there are two types of cursors, scrollable and basic, specifying the `rowlimit` input argument provides more control to manage memory. By specifying `rowlimit`, you can manage the number of rows to import at once instead of importing all rows from the resultset into MATLAB memory.

See Also

`exec` | `fetch` | `runsqlscript` | `setdbprefs`

Related Examples

- “Import Data from Databases into MATLAB” on page 6-4
- “Import Large Data Using Paging” on page 6-69

More About

- “Importing Data Using a Scrollable Cursor” on page 6-59
- “Working with Large Data Sets” on page 2-199

- "Preference Settings for Large Data Import" on page 5-19

Import Data Incrementally Using the Cursor Object

This example shows how to work with large data sets by retrieving data incrementally to avoid Java heap errors.

Create the Database Connection

Create a database connection `conn` to the Microsoft Access database using the native ODBC interface. For example, the following code assumes that you are connecting to a data source named `dbtoolboxdemo` with `admin` as the user name and password.

```
conn = database.ODBCConnection('dbtoolboxdemo','admin','admin');
```

Import Data in Batches

Use `fetch` with the `setdbprefs` properties for `FetchInBatches` and `FetchBatchSize` to import large data sets. Select data from the `productTable` table. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
setdbprefs('FetchInBatches','yes')
setdbprefs('FetchBatchSize','2')

curs = exec(conn,'select * from productTable');
curs = fetch(curs);
A = curs.Data
```

```
A =

    [ 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'
```

`fetch` internally retrieves data in increments of two rows at a time. Tune the `FetchBatchSize` setting depending on the size of the resultset you expect to import. For example, if you expect about 100,000 rows in the output, a batch size of 10,000 is a good starting point. The larger the `FetchBatchSize` value, the fewer trips between Java and

MATLAB, and the memory consumption is greater for each batch. The optimal value for FetchBatchSize is based on factors such as the:

- Size per row being retrieved
- Java heap memory value
- Default fetch size of the driver
- System architecture

Hence, the optimal value can vary across sites.

If 'FetchInBatches' is set to 'yes' and the total number of rows fetched is less than 'FetchBatchSize', MATLAB shows a warning message and then imports all the rows. The message is: Batch size specified was larger than the number of rows fetched.

Import Data Using a Row Limit

You can set a row limit on the final output even when the FetchInBatches setting is 'yes'.

```
setdbprefs('FetchInBatches','yes')
setdbprefs('FetchBatchSize','2')

curs = exec(conn,'select * from productTable');
curs = fetch(curs,3);
A = curs.Data
```

```
A =

    [9]    [125970]    [1003]    [13]    'Victorian Doll'
    [8]    [212569]    [1001]    [ 5]    'Train Set'
    [7]    [389123]    [1007]    [16]    'Engine Kit'
```

In this case, fetch retrieves the first three rows of productTable, two rows at a time.

Close the Cursor Object

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

```
close(curs)
```

See Also

database | exec | fetch | setdbprefs

Related Examples

- "Import Data from Databases into MATLAB" on page 6-4

More About

- "Connecting to a Database Using the Native ODBC Interface" on page 3-18
- "Preference Settings for Large Data Import" on page 5-19

Display Information About Imported Data

This example shows how to import data and display information about the imported data using a cursor object.

Alternatively, you can retrieve metadata of cursor objects by creating resultset objects using `resultset`. Display information about resultset objects using `rsmd`. Here, metadata refers to the information about the cursor object that contains the imported data after running `exec`.

Create the Database Connection and Import Data

Create the database connection `conn` using the `dbtoolboxdemo` data source. `dbtoolboxdemo` contains the table `productTable` with the column `productDescription`.

```
conn = database('dbtoolboxdemo','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 the Number of Rows in the Imported Data

Retrieve the number of rows `numrows` using `curs`.

```
numrows = rows(curs)

numrows =

    10
```

Retrieve the Number of Columns in the Imported Data

Retrieve the number of columns `numcols` using `curs`.

```
numcols = cols(curs)

numcols =

     1
```

Retrieve the Column Name in the Imported Data

Retrieve the column name `colname` using `curs`.

```
colname = columnnames(curs)

colname =

 'productDescription'
```

Retrieve the Column Width in the 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 the 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 the 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

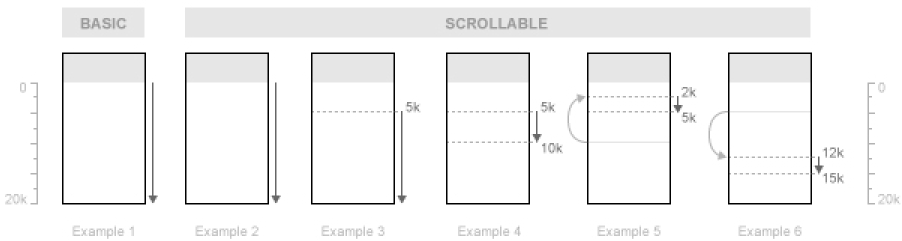
- "Import Data from Databases into MATLAB" on page 6-4

Importing Data Using a Scrollable Cursor

About 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 `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	`'absolutePo`	1	After the result set	Retrieves all rows in the cursor starting from the first row in the data set
Any	2	`'absolutePo`	1	1	Retrieves two rows in the cursor starting from the first row in the data set
Any	2	`'absolutePo`	5	5	Retrieves two rows in the cursor starting from the fifth row in the data set
Any	3	`'absolutePo`	-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	`'relativePo`	1	After the result set	Retrieves all rows in

Initial Scrollable Cursor Position	Row Limit	Scrollable Cursor Position Type	Offset	Ending Scrollable Cursor Position	`fetch` Action
					the cursor starting from the first row in the data set
Before result set	Any	`'relativePo`	Any	Varies	Retrieving with a relative position that starts before the result set causes behavior to vary based on the driver
5	2	`'relativePo`	5	10	Retrieves two rows in the cursor starting from the tenth row in the data set
11	3	`'relativePo`	-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	`'absolutePo`	1	0	Retrieves all rows in the cursor starting from the first row in the data set
Any	2	`'absolutePo`	1	2	Retrieves two rows in the cursor starting from the first row in the data set
Any	2	`'absolutePo`	5	6	Retrieves two rows in the cursor starting from the fifth row in the data set
Any	3	`'absolutePo`	-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	`'relativePo`	1	0	Retrieves all rows in the cursor starting from the first row in the data set
0	2	`'relativePo`	1	2	Retrieves the first two rows in the data set
5	2	`'relativePo`	5	11	Retrieves two rows in the data set starting from five rows from the initial position of five, which is nine
11	3	`'relativePo`	-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 6-4

More About

- "Import Data Using a Scrollable Cursor with a Relative Position Offset" on page 6-66

Import Data Using a Scrollable Cursor with a 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 the Database

Connect to the MySQL database using the native ODBC interface. This code assumes you are connecting to a data source named `MySQL` with user name `username` and password `pwd`.

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

Create a 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'`.

```
curs = exec(conn,'select * from productTable order by productNumber',...
            'cursorType','scrollable');
```

Retrieve Data Using an 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 a 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 the 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 6-4

More About

- "Importing Data Using a Scrollable Cursor" on page 6-59

Import Large Data Using Paging

Paging is a form of memory management that retrieves data from the database server in pages. Each page has a specified number of rows as a batch.

You can take advantage of paging by using `exec` with the name-value pair argument `'maxRows'` and the OFFSET syntax in the SQL query. Using `'maxRows'` alone, `exec` always retrieves data from the beginning of the data set. With the OFFSET syntax, you can reposition data retrieval to the next batch in the data set instead of the beginning. When working with large data, determine the correct values for the maximum number of rows and the OFFSET syntax. To determine these values, see “Preference Settings for Large Data Import” on page 5-19. The OFFSET syntax is different depending on the database. For details, consult your database documentation.

This example assumes that you have established a connection to a Microsoft SQL Server database named `conn`.

Define a helper function `build_query_with_offset` that concatenates the OFFSET syntax to the SQL query. This syntax is specific to the Microsoft SQL Server database. `query` is the SQL query. `offset` is the offset value.

```
function query = build_query_with_offset(query,offset)
    query = [query ' OFFSET ' num2str(offset) ' ROWS'];
end
```

Define the batch size to be 100,000 rows. Initialize the offset to zero. To store the resulting data from the query, initialize the variable `data`. Calculate the total number of batches `total_batches`. Define the SQL query `sqlquery` that selects the product description from the table `largedata`.

```
batchsize = 100000;
offset = 0;
data = {};

total_rows = fetch(conn,'SELECT COUNT(*) FROM largedata');
total_batches = total_rows{1} / batchsize;

sqlquery = 'SELECT productDescription FROM largedata ORDER BY productNumber';
```

Import the data one batch at a time using the offset value with the name-value pair argument `'maxRows'` in `exec`. Increment the offset value by the batch size.

```
for i = 1:total_batches
```

```
  % Build query with Offset (specific to each database)
  query = build_query_with_offset(sqlquery,offset);

  % Execute query with maxRows as batchsize
  curs = exec(conn,query,'maxRows',batchsize);

  % Fetch data
  curs = fetch(curs);

  % Store data in a variable
  data{i} = curs.Data;

  % Increment offset to new value
  offset = offset + batchsize;

end
```

The cell array `data` contains the resulting data.

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

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

See Also

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

More About

- "Managing Memory to Import Data" on page 6-50

Import Data Using a DatabaseDatastore Object

This example shows how to import data into MATLAB® using a DatabaseDatastore object. To access collections of data stored in a relational database, you can use a DatabaseDatastore object. 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. For an example of using a DatabaseDatastore object with tall arrays, see “Analyze Large Data in Database Using Tall Arrays”. Alternatively, you can write a MapReduce algorithm that defines the chunking and reduction of the data. For an example, see “Analyze Large Data in Database Using MapReduce”. For more MapReduce examples, see “Build Effective Algorithms with MapReduce”.

Create DatabaseDatastore Object

Using a JDBC driver, create a database connection conn to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to a database toy_store, a 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 dbds using the database connection conn and SQL query sqlquery. This SQL query retrieves all data from the table airlinesmall.

```
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       121
    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       181
```

Read the DatabaseDatastore object two more times by using the counter n. Read 10 records at a time.

```
n = 0;

while(hasdata(dbds) && n~=2)
    read(dbds)
    n = n+1;
end
```

```
ans =

    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRS
    ____    _____    __________    _________    _______    __________    _______    ___

    1987    10       16            5             959       1000          1212       121
    1987    10       17            6            2020       2020          2100       205
    1987    10        6            2            1132       1135          1426       141
    1987    10       24            6             944        945          1211       121
    1987    10       18            7             833        835          1003       101
    1987    10       26            1            2356       2355           730        72
    1987    10       29            4            1056       1055          1208       121
    1987    10        1            4            2304       2255          2340       232
    1987    10       30            5            1329       1329          1434       143
    1987    10        3            6            1040       1040          1125       112


ans =

    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRS
    ____    _____    __________    _________    _______    __________    _______    ___

    1987    10       23            5            1855       1855          2158       220
    1987    10       30            5            1055       1055          1302       131
    1987    10       28            3             NaN       1850           NaN       205
    1987    10       26            1            1600       1600          1649       164
    1987    10        6            2             745        745           833        82
    1987    10       31            6            1350       1350          1612       161
    1987    10       12            1            1253       1200          1359       131
    1987    10       19            1             650        645           852        81
    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 re-reading 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       121
    1987    10        9            5            1155       1155          1250       130
    1987    10       22            4             715        715           807        80
```

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

Import Data Using the 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 the MATLAB Interface to SQLite”. For more functionality, connect to the SQLite database file using the JDBC driver. For details, see “Configuring a Driver and Data Source”.

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,:)

suppliers_data(1:3,:)

salesVolume_data(1:3,:)

productTable_data(1:3,:)
```

```
ans =

  3×4 cell array

    [1]    [1700]    [14.5000]    '9/23/2014 9:38:3...'
    [2]    [1200]    [ 9.3000]    '7/8/2014 10:50:4...'
    [3]    [ 356]    [17.2000]    '5/14/2014 7:14:2...'


ans =

  3×5 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'


ans =

  3×13 cell array

  Columns 1 through 8

    [125970]    [1400]    [1100]    [ 981]    [ 882]    [794]    [752]    [654]
    [212569]    [2400]    [1721]    [1414]    [1191]    [983]    [825]    [731]
    [389123]    [1800]    [1200]    [ 890]    [ 670]    [550]    [450]    [400]

  Columns 9 through 13

    [773]    [809]    [980]    [3045]    [19000]
    [653]    [723]    [790]    [1400]    [ 5000]
    [410]    [402]    [450]    [1200]    [16000]


ans =

  3×5 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

- “Working with a Database and MATLAB” on page 2-3
- “Working with the MATLAB Interface to SQLite” on page 2-6
- “Configuring a Driver and Data Source” on page 2-16

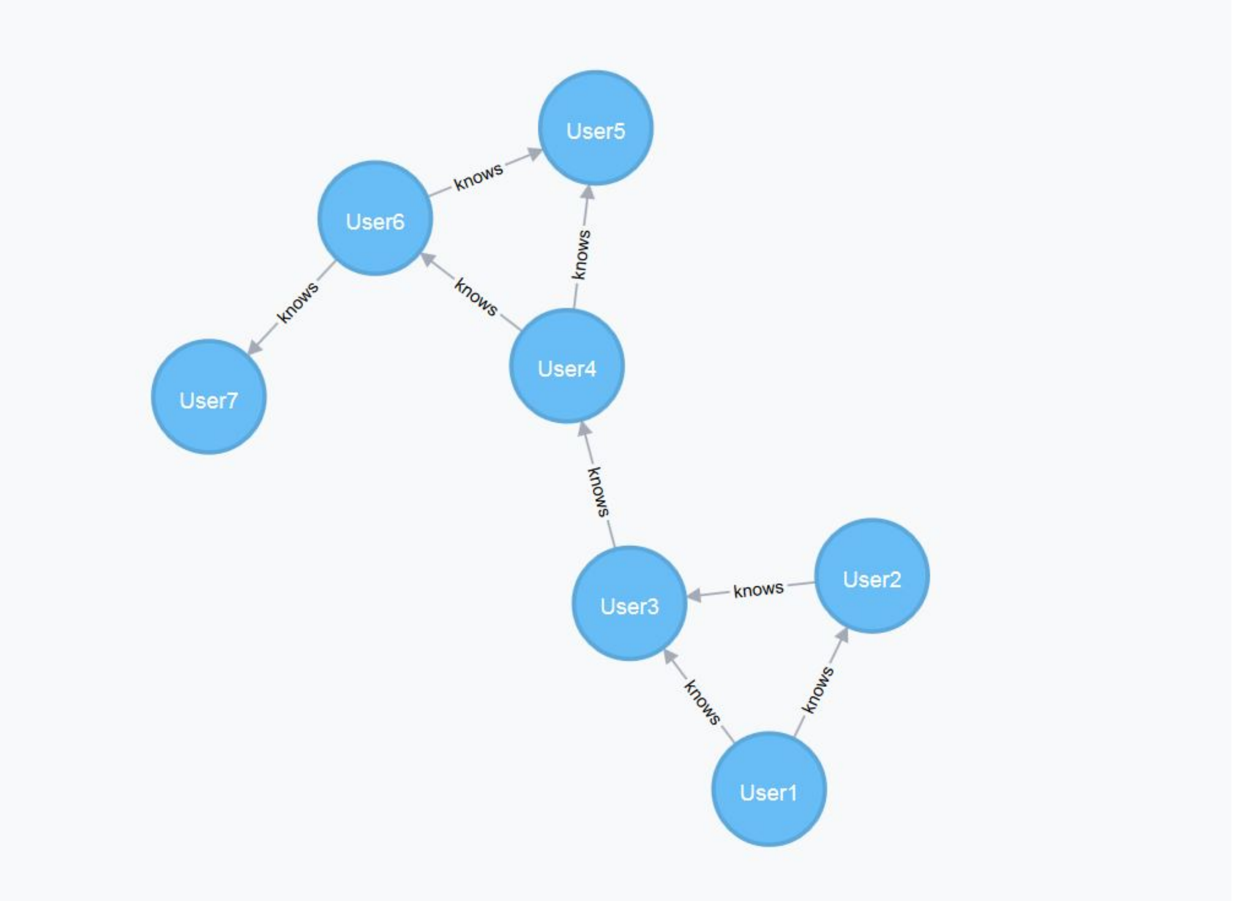
Neo4j Topics

Explore Graph Database Structure

This example shows how to traverse a graph and explore its structure using the MATLAB® interface to Neo4j®. For details about the MATLAB® interface to Neo4j®, see "Working with the MATLAB Interface to Neo4j".

Assume that you have graph data that is stored on a Neo4j® database which represents a social neighborhood. This database has seven nodes and eight relationships. Each node has only one unique property key `name` with values `User1` through `User7`. Each relationship has type `knows`.

The local machine hosts the Neo4j® database with port number `7474`, user name `neo4j`, and password `matlab`. For a visual representation of the data in the database, see this figure.



Connect to Neo4j® Database

Create a Neo4j® connection object `neo4jconn` using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j® connection object `neo4jconn`.

```
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 =

  cell

    'Person'
```

Find all the relationship types in the Neo4j® database.

```
reltypes = relationTypes(neo4jconn)
```

```
reltypes =

  cell

    'knows'
```

Find the property keys in the Neo4j® database.

```
propkeys = propertyKeys(neo4jconn)
```

```
propkeys =

  2×1 cell array
```

```
    'name'
    'property'
```

Search for Nodes

Search for all the nodes with the node label `Person`.

```
nlabel = 'Person';

nodesinfo = searchNode(neo4jconn,nlabel)
```

```
nodesinfo = 

         NodeLabels       NodeData                  NodeObject             
         __________    ____________    ________________________________

    0    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    1    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    2    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    3    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    4    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    5    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    6    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
```

`nodesinfo` contains node labels, node data, and the `Neo4jNode` objects for each matched node.

Search for the node that has the node identifier `2`.

```
nodeid = 2;

nodeinfo = searchNodeByID(neo4jconn,nodeid)
```

```
nodeinfo = 

  Neo4jNode with properties:

        NodeID: 2
      NodeData: [1×1 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 =

  cell

    '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 all incoming relationships that belong to the node `nodeinfo`.

```
relinfo = searchRelation(neo4jconn,nodeinfo,'in')
```

```
relinfo =

  struct with fields:

       Origin: 2
        Nodes: [2×3 table]
    Relations: [1×4 table]
```

`relinfo` contains data about the starting and ending 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: [7×3 table]
    Relations: [8×4 table]
```

`graphinfo` contains node data for all starting and ending 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`

More About

- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 7-10
- “Working with the MATLAB Interface to Neo4j” on page 7-8

Working with the MATLAB Interface to Neo4j

The MATLAB interface to Neo4j lets you connect to a Neo4j graph database and import graph data into MATLAB. You can perform graph network analysis by creating a directed graph from the imported graph data. Or, if you are familiar with the Cypher® query language, you can execute Cypher queries on the Neo4j database.

About Neo4j Graph Databases

A graph database stores data using a graph data model. This model consists of nodes and relationships. A relationship describes how two or more nodes are related to each other.

Nodes can have zero or more node 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 and 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". 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 the graph database, 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`
- `nodeDegree`
- `nodeRelationTypes`

3 To perform graph network analysis, you can convert output structures to `digraph` objects using `neo4jStruct2Digraph`. For details, see "Directed and Undirected Graphs".

   Or, if you know the Cypher query language, you can execute a Cypher query using `executeCypher`. For details, see Cypher Query Language.

See Also
`digraph` | `neo4j` | `neo4jStruct2Digraph`

Related Examples
- "Find Friends of Friends in Social Neighborhood"
- "Visualize Breadth-First and Depth-First Search"

More About
- "Directed and Undirected Graphs"
- "Searching Graph Database Using MATLAB Interface to Neo4j" on page 7-10
- "MATLAB Interface to Neo4j Error Messages" on page 7-13

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".

   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`.

   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` using `neo4jStruct2Digraph`.

```
G = neo4jStruct2Digraph(relinfo);
```

4 Perform graph network analysis using the `digraph` object `G`. For details, see "Directed and Undirected Graphs".

   For example, determine the shortest path between nodes using `distances`.

```
d = distances(G);
```

   Or, explore the graph data by accessing the output structures `nodeinfo` and `relinfo`.

See Also

`nodeDegree` | `searchGraph` | `searchNode` | `searchNodeByID` | `searchRelation`

Related Examples

- "Explore Graph Database Structure" on page 7-2

- “Find Shortest Path Between People in Social Neighborhood”

More About

- “Working with the MATLAB Interface to Neo4j” on page 7-8
- “MATLAB Interface to Neo4j Error Messages” on page 7-13

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 `neo4j`.
Unable to connect. Please try again.	The Neo4j database connection is invalid.	Connect to the Neo4j database using `neo4j`.
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 `searchNode`.
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 `searchRelation`.
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 `nodeLabels`.
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 `relationTypes`.
No property keys found.	The Neo4j database has no property keys.	Open the Neo4j database and add property keys.

Error Message	Probable Causes	Resolution
		For details, see the Neo4j Operations Manual in the Neo4j Documentation. Then, run `propertyKeys`.
Unable to execute Cypher query.	The Cypher query is invalid.	Verify the Cypher query. Then, run `executeCypher`. For details about writing Cypher queries, see Cypher Query Language.

See Also
`neo4j` | `searchGraph`

Related Examples
- "Find Friends of Friends in Social Neighborhood"
- "Explore Graph Database Structure" on page 7-2

More About
- "Working with the MATLAB Interface to Neo4j" on page 7-8
- "Searching Graph Database Using MATLAB Interface to Neo4j" on page 7-10

External Websites
- Neo4j Documentation
- Cypher Query Language

Functions — Alphabetical List

attr

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);
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

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);
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

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 6-56

Input Arguments

`curs` — Database cursor

database cursor object

Database cursor, specified as an open SQL database cursor object created using `exec`.

`colnum` — Column number

scalar

Column number, specified as a 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
`fieldName`	Name of the column.
`typeName`	Data type.
`typeValue`	Numerical representation of the data type.
`columnWidth`	Size of the field.
`precision`	Precision value for floating and double data types; an empty value is returned for character vectors.
`scale`	Precision value for real and numeric data types; an empty value is returned for character vectors.

Attribute	Description
currency	If this equals true, the data format is currency.
readOnly	If this equals true, the data cannot be overwritten.
nullable	If this equals true, the data can be NULL.
Message	Error message returned by fetch.

See Also

close | cols | columnnames | columns | database | dmd | fetch | get | tables | width

Introduced before R2006a

bestrowid

(Not recommended) Unique identifier for row in database table

Compatibility

`bestrowid` has been removed.

Syntax

```
b = bestrowid(dbmeta,'cata','sch')
b = bestrowid(dbmeta,'cata','sch','tab')
```

Description

`b = bestrowid(dbmeta,'cata','sch')` returns the optimal set of columns in a table that uniquely identifies a row in the schema `sch`, in the catalog `cata`, for the database whose database metadata object is `dbmeta`.

`b = bestrowid(dbmeta,'cata','sch','tab')` returns the optimal set of columns that uniquely identifies a row in table `tab`, in the schema `sch`, in the catalog `cata`, for the database whose database metadata object is `dbmeta`.

Examples

Run `bestrowid`, passing it the following arguments:

- `dbmeta`, the database metadata object
- `msdb`, the catalog
- `geck`, the schema
- `builds`, the table

```
b = bestrowid(dbmeta, 'msdb', 'geck', 'builds')
b =
```

```
    'build_id'
```

The result indicates that each entry in the build_id column is unique and identifies the row.

See Also

columns | dmd | get | tables

Introduced before R2006a

catalogs

Get database catalog names

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 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 6-39

Input Arguments

`conn` — Database connection
database connection object

Database connection, specified as a database connection object created using `database`.

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`

Introduced in R2010a

clearwarnings

(Not recommended) Clear warnings for database connection or resultset

Compatibility

`clearwarnings` has been removed.

Syntax

```
clearwarnings(conn)
clearwarnings(rset)
```

Description

`clearwarnings(conn)` clears warnings reported for the database connection object `conn`.

`clearwarnings(rset)` clears warnings reported for the `resultset` object `rset`.

Tip For command-line help on `clearwarnings`, use the overloaded methods:

```
help database/clearwarnings
help resultset/clearwarnings
```

See Also

`database` | `get` | `resultset`

Introduced before R2006a

close

Close database and driver resource utilizer

Compatibility

`resultset` will be removed in a future release.

Syntax

```
close(object)
```

Description

`close(object)` closes the database and driver resource utilizer `object` to free up database and driver resources.

Examples

Close Database Connection Object

Using the native ODBC interface, connect to the database with the ODBC data source name `dbtoolboxdemo`, the user name `admin`, and the password `admin`.

```
conn = database.ODBCConnection('dbtoolboxdemo','admin','admin');
```

Close the database connection `conn`.

```
close(conn)
```

Close SQLite Connection Object

Create a SQLite connection `conn` 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);
```

Close the SQLite connection `conn`.

```
close(conn)
```

Close `DatabaseDatastore` Object

Using a JDBC driver, create a database connection `conn` to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to a database `toy_store`, a 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 `dbds` using the database connection `conn` and SQL query `sqlquery`. This SQL query retrieves all data from the table `airlinesmall`.

```
sqlquery = 'select * from airlinesmall';
```

```
dbds = databaseDatastore(conn,sqlquery);
```

Close the `DatabaseDatastore` object `dbds`.

```
close(dbds)
```

Close Cursor Object

Using the native ODBC interface, connect to the database with the ODBC data source name `dbtoolboxdemo`, the user name `admin`, and the password `admin`.

```
conn = database.ODBCConnection('dbtoolboxdemo','admin','admin');
```

Select data from the table `productTable` that you access using the `database.ODBCConnection` object `conn`. Assign the returned cursor object to the variable `curs`.

```
sqlquery = 'select * from productTable';
curs = exec(conn,sqlquery);
```

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

```
close(curs)
```

Close the database connection `conn`.

```
close(conn)
```

Close Resultset Object

Connect to the database with the ODBC data source name `dbtoolboxdemo`, the user name `admin`, and the password `admin`.

```
conn = database('dbtoolboxdemo','admin','admin');
```

Select data from the table `productTable` that you access using the database connection object `conn`. Assign the returned cursor object to the variable `curs`.

```
sqlquery = 'select * from productTable';
curs = exec(conn,sqlquery);
```

Construct a resultset object `rset`.

```
rset = resultset(curs);
```

Close the resultset object `rset`.

```
close(rset)
```

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

```
close(curs)
```

Close the database connection `conn`.

```
close(conn)
```


- "Import Data from Databases into MATLAB" on page 6-4
- "Import Data Using a `DatabaseDatastore` Object" on page 6-71
- "Export Data to New Record in Database" on page 6-22
- "Display Information About Imported Data" on page 6-56
- "Import Data Using the MATLAB® Interface to SQLite" on page 6-75

Input Arguments

`object` — Database and driver resource utilizer

database connection object | SQLite connection object | `DatabaseDatastore` object | cursor object | resultset object

Database and driver resource utilizer, specified as one of these objects.

Object Argument Name	Object Name	Object Description	Object Creation Function
`conn`	Database connection object or native ODBC database connection object	A database connection object creates a connection between an installed database and MATLAB. For details, see "Connecting to a Database" on page 2-191.	`database`
`conn`	SQLite connection object	A SQLite connection object creates a connection to a SQLite database file using the MATLAB interface to SQLite. For details, see "Working with the MATLAB Interface to SQLite" on page 2-6.	`sqlite`
`dbds`	`DatabaseDatastore` object	A `DatabaseDatastore` object creates a connection to a type of datastore for working with large data.	`databaseDatastore`
`curs`	Cursor object or native ODBC cursor object	A cursor object stores imported data.	`exec`
`rset`	Resultset object	A resultset object provides metadata about cursor objects.	`resultset`

Database connections, SQLite connections, `DatabaseDatastore` objects, cursors, and resultset objects remain open until you close them using the `close` function. Always close a cursor, `DatabaseDatastore`, database or SQLite connection, or resultset object when you finish using it. Close a cursor before closing the connection used for that cursor.

Executing `close` with a `DatabaseDatastore` object releases the MATLAB resources associated with database connection and cursor objects.

Note: The MATLAB session closes open cursors, `DatabaseDatastore` objects, and connections when exiting. However, the database might not free up the cursors and connections.

More About

- "Configuring a Driver and Data Source" on page 2-16
- "Connecting to a Database" on page 2-191
- DatabaseDatastore
- "Working with the MATLAB Interface to SQLite" on page 2-6

See Also

`database` | `databaseDatastore` | `exec` | `fetch` | `resultset` | `sqlite`

Introduced before R2006a

cols

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 `dbtoolboxdemo` data source.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Working with the `dbtoolboxdemo` 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` element 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` element 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 6-56

Input Arguments

`curs` — Database cursor

database cursor object

Database cursor, specified as an open SQL database cursor object created using `exec`.

Output Arguments

`numcols` — Number of columns

scalar

Number of columns in a data set, returned as a scalar.

More About

- “Connecting to a Database Using the Native ODBC Interface” on page 3-18

See Also

`attr` | `close` | `columnnames` | `columnprivileges` | `columns` | `database` | `fetch` | `get` | `rows` | `width`

Introduced before R2006a

columnnames

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 `dbtoolboxdemo` data source.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Working with the `dbtoolboxdemo` 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 `dbtoolboxdemo` data source.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Working with the `dbtoolboxdemo` 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 6-56

Input Arguments

`curs` — Database cursor
database cursor object

Database cursor, specified as an open SQL database cursor object created using `exec`.

`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`

Introduced before R2006a

columnprivileges

List database column privileges

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`

Introduced before R2006a

columns

Return database table column names

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 the Column List for a Catalog Using the Database Connection

Create a database connection `conn` using the native ODBC interface to the Microsoft SQL Server database. For example, the following code assumes that you are connecting to a data source named `MS SQL Server` with user name `username` and password `pwd`.

```
conn = database.ODBCConnection('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 the Column List for a Catalog and Schema Using the Database Connection

Create a database connection `conn` using the native ODBC interface to the Microsoft SQL Server database. For example, the following code assumes that you are connecting to a data source named `MS SQL Server` with user name `username` and password `pwd`.

```
conn = database.ODBCConnection('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 the Column List for a Catalog, Schema, and Table Name Using the Database Connection

Create a database connection `conn` using the native ODBC interface to the Microsoft SQL Server database. For example, the following code assumes that you are connecting to a data source named `MS SQL Server` with user name `username` and password `pwd`.

```
conn = database.ODBCConnection('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 the Column List for a Catalog Using the 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 the Column List for a Catalog and Schema Using the 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 the Column List for a Catalog, Schema, and Table Name Using the 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 6-39

Input Arguments

`conn` — Database connection
database connection object

Database connection, specified as a database connection object created using `database`.

`dbmeta` — Database metadata
database metadata object

Database metadata, specified as a database metadata object created using `dmd`. To use this object, connect to the database using the JDBC driver or JDBC/ODBC bridge. For details about database connections, see `database`.

`catalog` — Database catalog name
character vector

Database catalog name, specified as a character vector.

Data Types: `char`

`schema` — Database schema name
character vector

Database schema name, specified as a character vector.

Data Types: `char`

`tablename` — Database table name
character vector

Database table name, specified as a character vector denoting the name of a table in your database.

Data Types: `char`

Output Arguments

`columnlist` — List of columns
cell array

List of columns, returned as a cell array.

More About

- "Connecting to a Database Using the Native ODBC Interface" on page 3-18

See Also

`attr` | `close` | `cols` | `columnnames` | `columnprivileges` | `database` | `dmd` | `get`

Introduced in R2010a

commit

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`.

```
get(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)
```

More About

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`

Introduced before R2006a

confds

Configure JDBC data source for Visual Query Builder

Alternatives

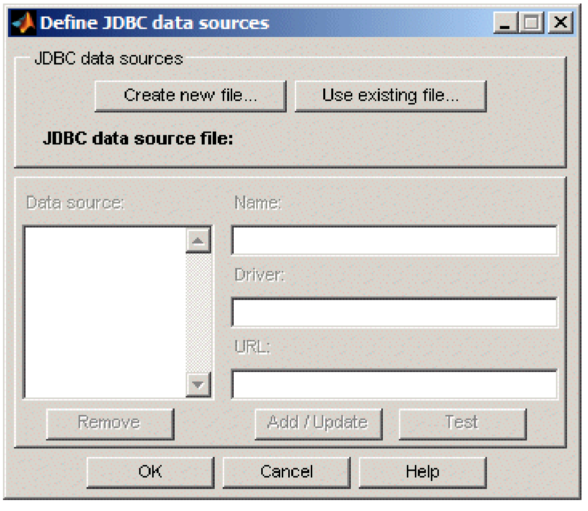
Select **Define JDBC data sources** from the Visual Query Builder **Query** menu.

Syntax

```
confds
```

Description

`confds` displays the VQB Define JDBC data sources dialog box. Use `confds` only to build and run queries using Visual Query Builder with JDBC drivers.



For information about how to use the Define JDBC data sources dialog box to configure JDBC drivers, see “Configuring a Driver and Data Source” on page 2-16.

Tip Use the `database` function to define JDBC data sources programmatically.

See Also
`database` | `querybuilder`

Introduced before R2006a

crossreference

(Not recommended) Retrieve information about primary and foreign keys

Compatibility

crossreference has been removed.

Syntax

```
f = crossreference(dbmeta, 'pcata', 'psch', 'ptab', 'fcata', 'fsch',
'ftab')
```

Description

f = crossreference(dbmeta, 'pcata', 'psch', 'ptab', 'fcata', 'fsch', 'ftab') returns information about the relationship between foreign keys and primary keys for the database whose database metadata object is dbmeta. The primary key information is for the table ptab in the primary schema psch. The primary catalog is pcata. The foreign key information is for the foreign table ftab in the foreign schema fsch. The foreign catalog is fcata.

Examples

Run crossreference to get primary and foreign key information. The database metadata object is dbmeta, the primary and foreign catalog is orcl, the primary and foreign schema is SCOTT, the table that contains the referenced primary key is DEPT, and the table that contains the foreign key is EMP.

```
f = crossreference(dbmeta,'orcl','SCOTT','DEPT',...
 'orcl','SCOTT','EMP')
f = 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 following primary and foreign key information.

Column	Description	Value
1	Catalog that contains primary key, referenced by foreign imported key	`orcl`
2	Schema that contains primary key, referenced by foreign imported key	`SCOTT`
3	Table that contains 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 key	`orcl`
6	Schema that has foreign key	`SCOTT`
7	Table that has foreign key	`EMP`
8	Foreign key 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`

There is only one foreign key in the schema `SCOTT`. The table `DEPT` contains a primary key `DEPTNO` that is referenced by the field `DEPTNO` in the table `EMP`. The field `DEPTNO` in the `EMP` table is a foreign key.

Tip For a description of the codes for update and delete rules, see the `getCrossReference` property on the Oracle Java Web site:

`http://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html`.

See Also

`dmd` | `exportedkeys` | `get` | `importedkeys` | `primarykeys`

Introduced before R2006a

database

Connect to database

There are three ways to connect to a database. For ODBC drivers, connect to a database using the native ODBC interface or the JDBC/ODBC bridge. For JDBC drivers, connect to a database using a JDBC driver. For details about deciding which connection option is best in your situation, see "Choosing Between ODBC and JDBC Drivers" on page 2-13. For details about the native ODBC interface, see "Connecting to a Database Using the Native ODBC Interface" on page 3-18.

Note: The JDBC/ODBC bridge functionality will be removed in a future release. To connect to a database, use the native ODBC interface or a JDBC driver instead.

The `database` function creates a database connection object. You can use this object to connect to various databases using different drivers that you install and administer. For details, see "Connecting to a Database" on page 2-191.

A database connection object is one of the two available database connection types. The other creates a SQLite database connection object using the function `sqlite`. This object lets you connect to a SQLite database file using the MATLAB interface to SQLite without installing or administering a database or driver. For details, see "Working with the MATLAB Interface to SQLite" on page 2-6.

Syntax

```
conn = database.ODBCConnection(instance,username,password)
conn = database(instance,username,password)

conn = database(instance,username,password,driver,databaseurl)

conn = database(instance,username,password,Name,Value)
```

Description

`conn = database.ODBCConnection(instance,username,password)` returns a database connection object for the connection to the ODBC data source setup `instance` using the native ODBC interface.

`conn = database(instance,username,password)` returns a database connection object for the connection to the ODBC data source setup `instance` using the JDBC/ODBC bridge.

`conn = database(instance,username,password,driver,databaseurl)` connects to the database `instance` using a JDBC driver.

`conn = database(instance,username,password,Name,Value)` connects to the database `instance` using a JDBC driver with connection properties specified by one or more `Name,Value` pair arguments.

Examples

Connect to Microsoft Access Using the Native ODBC Interface

Connect to the database with the ODBC data source name `dbtoolboxdemo` using the user name `username` and password `pwd`.

```
conn = database.ODBCConnection('dbtoolboxdemo','username','pwd')
```

```
conn =

  connection with properties:

      Instance: 'dbtoolboxdemo'
      UserName: 'username'
       Message: []
        Handle: [1x1 database.internal.ODBCConnectHandle]
       TimeOut: 0
    AutoCommit: 0
          Type: 'ODBCConnection Object'
```

`database.ODBCConnection` returns `conn` as `database.ODBCConnection` object. `conn` has an empty `Message` property, which indicates a successful connection.

Close the database connection `conn`.

```
close(conn)
```

Connect to Microsoft Access Using the JDBC/ODBC Bridge

Connect to the database with the ODBC data source name `dbtoolboxdemo` using the user name `username` and password `pwd`.

```
conn = database('dbtoolboxdemo','username','pwd')
```

```
conn =

  connection with properties:

       Instance: 'dbtoolboxdemo'
       UserName: 'username'
         Driver: []
            URL: []
    Constructor: [1x1 com.mathworks.toolbox.database.databaseConnect]
        Message: []
         Handle: [1x1 sun.jdbc.odbc.JdbcOdbcConnection]
        TimeOut: 0
     AutoCommit: 'on'
           Type: 'Database Object'
```

`database` returns `conn` as a `Database Object`. `conn` has an empty `Message` property, which indicates a successful connection.

Close the database connection `conn`.

```
close(conn)
```

Connect to Microsoft SQL Server Using Windows Authentication

Connect to a Microsoft SQL Server database with integrated Windows Authentication using a JDBC driver.

Use the `AuthType` parameter to establish a Windows Authentication connection. For details about how to set up Windows Authentication and find your port number, see “Microsoft SQL Server JDBC for Windows” on page 2-35.

```
conn = database('test_db','','',...
   'Vendor','Microsoft SQL Server','Server','servername',...
   'AuthType','Windows','portnumber',123456)
```

```
conn =

  connection with properties:

       Instance: 'test_db'
       UserName: ''
         Driver: []
            URL: []
    Constructor: [1x1 com.mathworks.toolbox.database.databaseConnect]
        Message: []
         Handle: [1x1 com.microsoft.sqlserver.jdbc.SQLServerConnection]
        TimeOut: 0
     AutoCommit: 'on'
           Type: 'Database Object'
```

`conn` has an empty `Message` property, which indicates a successful connection.

Close the database connection `conn`.

```
close(conn)
```

Connect to Sybase Using a JDBC Driver URL

Connect to the database `dbname` using the user name `username` and password `pwd`. Use the JDBC driver `com.sybase.jdbc4.jdbc.SybDriver` to make the connection. Use the URL defined by the driver vendor including your server name, port number, and database name. For details, see "Sybase JDBC for Windows" on page 2-97.

```
conn = database('dbname','username','pwd',...
                'com.sybase.jdbc4.jdbc.SybDriver','URL')
```

```
conn =

  connection with properties:

       Instance: 'dbname'
       UserName: 'username'
         Driver: 'com.sybase.jdbc4.jdbc.SybDriver'
            URL: 'URL'
    Constructor: [1x1 com.mathworks.toolbox.database.databaseConnect]
        Message: []
         Handle: [1x1 com.sybase.jdbc4.jdbc.SybConnection]
        TimeOut: 0
     AutoCommit: 'on'
           Type: 'Database Object'
```

`conn` has an empty `Message` property, which indicates a successful connection.

Close the database connection `conn`.

```
close(conn)
```

Connect to Oracle Using a JDBC Driver

Connect to an Oracle database using name-value pair arguments to specify the vendor and connection options.

Connect to the database `test_db` using the user name `username` and password `pwd`. Enter the driver type as `thin` for a default connection to Oracle. To connect to Oracle with Windows authentication, use `oci`. The database server machine name is `remotehost` and the port number that the server is listening on is `1234`. For details, see "Oracle JDBC for Windows" on page 2-50.

```
conn = database('test_db','username','pwd','Vendor','Oracle',...
          'DriverType','thin','Server','remotehost','PortNumber',1234)
```

```
conn =

  connection with properties:

       Instance: 'test_db'
       UserName: 'username'
         Driver: []
            URL: []
    Constructor: [1x1 com.mathworks.toolbox.database.databaseConnect]
        Message: []
         Handle: [1x1 oracle.jdbc.driver.T4CConnection]
        TimeOut: 0
     AutoCommit: 'on'
           Type: 'Database Object'
```

`conn` has an empty `Message` property, which indicates a successful connection.

Close the database connection `conn`.

```
close(conn)
```

Connect to MySQL Using a JDBC Driver

Connect to a MySQL database using name-value pair arguments to specify the vendor and connection options.

Connect to the database test_db on the machine remotehost. Use the user name username and password pwd. For details, see "MySQL JDBC for Windows" on page 2-65.

```
conn = database('test_db','username','pwd','Vendor','MySQL',...
                'Server','remotehost')

conn =

   connection with properties:

      Instance: 'test_db'
       UserName: 'username'
         Driver: []
            URL: []
    Constructor: [1x1 com.mathworks.toolbox.database.databaseConnect]
        Message: []
         Handle: [1x1 com.mysql.jdbc.JDBC4Connection]
        TimeOut: 0
     AutoCommit: 'on'
           Type: 'Database Object'
```

conn has an empty Message property, which indicates a successful connection.

Close the database connection conn.

```
close(conn)
```

Connect to Microsoft Access Using a File DSN

Connect to a Microsoft Access database with .accdb format using an OBDC driver.

Specify the location of the database on the disk.

```
dbpath = ['C:\Data\Matlab\MyDatabase.accdb'];
```

Create the connection URL.

```
url = [['jdbc:odbc:Driver={Microsoft Access Driver (*.mdb, *.accdb)};DSN='''';DBQ='] dbpath];
```

Connect to the database MyDatabase.accdb using dpath and url.

```
conn = database('','','','sun.jdbc.odbc.JdbcOdbcDriver',url);
```

Fetch data from the database.

```
curs = exec(conn,'SELECT ALL January FROM salesVolume');
curs = fetch(curs);
```

```
data = curs.Data;
```

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

```
close(curs)
```

Close the database connection `conn`.

```
close(conn)
```

Connect to PostgreSQL Using a JDBC Driver

Connect to a PostgreSQL database using name-value pair arguments to specify the vendor and connection options.

Connect to the database `test_db` using the user name `username` and password `pwd` on the machine `remotehost`. For details, see "PostgreSQL JDBC for Windows" on page 2-77.

```
conn = database('test_db','username','pwd','Vendor','PostgreSQL',...
                'Server','remotehost')
```

```
conn =

  connection with properties:

       Instance: 'test_db'
       UserName: 'username'
         Driver: []
            URL: []
    Constructor: [1x1 com.mathworks.toolbox.database.databaseConnect]
        Message: []
         Handle: [1x1 org.postgresql.jdbc4.Jdbc4Connection]
        TimeOut: 0
     AutoCommit: 'on'
           Type: 'Database Object'
```

`conn` has an empty `Message` property, which indicates a successful connection.

Close the database connection `conn`.

```
close(conn)
```

- "Import Data from Databases into MATLAB" on page 6-4

- "Export Data to New Record in Database" on page 6-22
- "Display Database Metadata" on page 6-39

Input Arguments

`instance` — Data source setup or database name
character vector

Data source setup or database name, specified as a character vector. Specify a data source for ODBC connection, and the database name for JDBC connection. For an ODBC driver, `instance` is the name you provide for your data source when you create a data source using the Microsoft ODBC Administrator. For a JDBC driver, `instance` is the name of your database. The name differs for different database systems. For example, `instance` is the SID or the service name when you are connecting to an Oracle database. Or, `instance` 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.

`username` — User name
character vector

User name required to access the database, specified as a character vector. If no user name is required, specify empty value `''`.

`password` — Password
character vector

Password required to access the database, specified as a character vector. If no password is required, specify empty value `''`.

`driver` — JDBC driver name
character vector

JDBC driver name, specified as a character vector 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.

`databaseurl` — Database connection URL
character vector

Database connection URL, specified as a character vector 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. If you do not know the driver name or the URL, you can use name-value pair arguments to specify individual connection properties.

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 on a machine named `remotehost`.

`'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 connecting to a database system not listed here, use the `driver` and `databaseurl` syntax.

Example: `'Vendor','Oracle'`

`'Server'` — Database server

`'localhost'` (default) | character vector

Database server name or address, specified as the comma-separated pair consisting of `'Server'` and a character vector.

Example: `'Server','remotehost'`

`'PortNumber'` — Server port

scalar

Server port number that the server is listening on, specified as the comma-separated pair consisting of `'PortNumber'` and a scalar value.

Example: `'PortNumber',1234`

Data Types: `double`

`'AuthType'` — Authentication
`'Server'` (default) | `'Windows'`

Authentication type (valid only for Microsoft SQL Server), specified as the comma-separated pair consisting of `'AuthType'` and one of these values:

- `'Server'`
- `'Windows'`

Specify `'Windows'` for 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'`
- `'oci'`

Specify `'oci'` for Windows Authentication.

Example: `'DriverType','thin'`

`'URL'` — Connection URL
character vector

Connection URL, specified as the comma-separated pair consisting of `'URL'` and a character vector. If you specify `URL`, do not specify the other name-value pair arguments.

Output Arguments

`conn` — Database connection
database connection object

Database connection, returned as a database connection object. The database connection object has the following properties.

Property	Description
`Instance`	Data source name when using ODBC or database name when using JDBC
`UserName`	User name used for database login
`Driver`	JDBC or JDBC/ODBC driver object used for database connection
`URL`	Driver vendor-specific string for database connection
`Constructor`	Internal Java or C++ representation of database connection object
`Message`	Database connection status message that is empty when a successful connection is established
`Handle`	Internal Java or C++ representation of database connection object
`TimeOut`	Number of seconds that the driver waits while trying to establish a database connection before throwing an error
`AutoCommit`	Set to `on` to apply updates to the database automatically and set to `off` to commit updates to the database manually
`Type`	Database connection object or `database.ODBCConnection` object

Depending on the database connection, the database connection object properties and property values vary. For details on the differences, see this table.

Database Connection Type	Database Connection Object Property Differences
Native ODBC database connection	• Excludes `Driver`, `URL`, and `Constructor` properties. • `Type` property is equal to `database.ODBCConnection` object. • `Handle` property is `database.internal.ODBCConnectHandle`.

Database Connection Type	Database Connection Object Property Differences
	• `Instance` property contains the data source name.
JDBC/ODBC bridge connection	• `Instance` property contains the data source name. • `Handle` property is `sun.jdbc.odbc.JdbcOdbcConnection`.
JDBC driver connection	`Instance` property contains the database name.

More About

JDBC Driver Name and Database Connection URL

The JDBC driver name and database connection URL take different forms for different databases, as shown in the following table.

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`
MySQL	**JDBC driver:** `twz1.jdbc.mysql.jdbcMysqlDriver` **Database URL:** `jdbc:z1MySQL://natasha:3306/metrics` **JDBC driver:** `com.mysql.jdbc.Driver` **Database URL:** `jdbc:mysql://devmetrics.mrkps.com/testing` To insert or select characters with encodings that are not default, append the value `useUnicode=true&characterEncoding=`*`encoding`* to the URL,

Database	JDBC Driver Name and Database URL Example Syntax
	where *encoding* is any valid MySQL character encoding followed by `&`. For example, `useUnicode=true&characterEncoding=utf8&`. *The trailing `&` is required.*
Oracle oci7 drivers	**JDBC driver:** `oracle.jdbc.driver.OracleDriver` **Database URL:** `jdbc:oracle:oci7:@rex`
Oracle oci8 drivers	**JDBC driver:** `oracle.jdbc.driver.OracleDriver` **Database URL:** `jdbc:oracle:oci8:@111.222.333.44:1521:` **Database URL:** `jdbc:oracle:oci8:@frug`
Oracle 10 Connections with JDBC (Thin drivers)	**JDBC driver:** `oracle.jdbc.driver.OracleDriver` **Database URL:** `jdbc:oracle:thin:`
Oracle Thin drivers	**JDBC driver:** `oracle.jdbc.driver.OracleDriver` **Database URL:** `jdbc:oracle:thin:@144.212.123.24:1822:` **Database URL:** `jdbc:oracle:thin:@(DESCRIPTION = (ADDRESS = (PROTOCOL = TCP)(HOST = ServerName)(PORT = 1234)) (CONNECT_DATA = (SERVER = DEDICATED) (SERVICE_NAME = dbname) ) )`
PostgreSQL	**JDBC driver:** `org.postgresql.Driver` **Database URL:** `jdbc:postgresql://host:port/database`
PostgreSQL with SSL Connection	**JDBC driver:** `org.postgresql.Driver` **Database URL:** `jdbc:postgresql:servername:dbname:ssl=true&sslfactory=org.postgresql.ssl.NonValidatingFactory&` *The trailing `&` is required.*
Sybase SQL Server and Sybase SQL Anywhere	**JDBC driver:** `com.sybase.jdbc4.jdbc.SybDriver` **Database URL:** `jdbc:sybase:Tds:yourhostname:yourportnumber/`

Tips

- Use `logintimeout` before `database` to set the maximum time for a connection attempt.
- Alternatively, connect to databases using Database Explorer.
- When making a JDBC connection using name-value connection properties:
  - You can skip the `Server` parameter when connecting to a database locally.
  - You can skip the `PortNumber` parameter when connecting to a database server listening on the default port (except for Oracle connections).

- "Initial Setup Requirements" on page 2-12
- "Connection Options" on page 2-9
- "Configuring a Driver and Data Source" on page 2-16
- "Connecting to a Database" on page 2-191
- "Connecting to a Database Using the Native ODBC Interface" on page 3-18
- "Database Connection Error Messages" on page 3-10
- "Bring Java Classes into MATLAB Workspace"
- "Inserting Data Using the Command Line" on page 2-197

See Also

Functions
`close` | `datainsert` | `dmd` | `exec` | `get` | `getdatasources` | `isopen` | `isreadonly` | `logintimeout` | `ping` | `querybuilder` | `supports` | `update`

Apps
Database Explorer

Introduced before R2006a

datastore

(Not recommended) Create datastore to access collection of data in a database

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”.

Compatibility

`datastore` is not recommended. Use `databaseDatastore` instead.

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 native ODBC interface. 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.ODBCConnection('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: [1×1 database.odbc.connection]
            Query: 'select * from productTable'
    VariableNames: {1×5 cell}
         ReadSize: 10000
```

`datastore` executes the SQL query `sqlquery` and creates a cursor object with the resulting data. `dbds` contains these properties:

- Database 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:

      Instance: 'MySQLdb'
      UserName: 'root'
       Message: []
        Handle: [1×1 database.internal.ODBCConnectHandle]
       TimeOut: 0
    AutoCommit: 'on'
          Type: 'ODBCConnection Object'
```

The `Message` property is blank when the database connection is successful.

Close the `DatabaseDatastore` and database connection.

```
close(dbds)
```

- "Import Data Using a `DatabaseDatastore` Object" on page 6-71

Input Arguments

`conn` — Database connection
database connection object

Database connection, specified as a database connection object created using `database`.

`sqlquery` — SQL statement
character vector

SQL statement, specified as a character vector.

Data Types: `char`

Output Arguments

`dbds` — Datastore containing data in database
`DatabaseDatastore` object

Datastore containing data in database, returned as a `DatabaseDatastore` object.

More About

- DatabaseDatastore
- "Getting Started with Datastore"

See Also

`close` | `database` | `databaseDatastore` | `datastore` | `preview` | `read`

Introduced in R2014b

hasdata

Determine if data in `DatabaseDatastore` is available to read

Compatibility

If there is no more data to read from the query, `hasdata` returns logical `0`.

Syntax

```
tf = hasdata(dbds)
```

Description

`tf = hasdata(dbds)` returns logical `1` (`true`) if there is data available to read from the `DatabaseDatastore` object `dbds`. Otherwise, it returns logical `0` (`false`).

Examples

Determine If `DatabaseDatastore` Object Contains More Data

Using a JDBC driver, create a database connection `conn` to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to a database `toy_store`, a 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 `dbds` using the database connection `conn` and SQL query `sqlquery`. This SQL query reads the first 30 rows of data from the table `airlinesmall`.

```
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        81
    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 `dbds` 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       134
    1990    9        12            3            1125       1125          1231       123
    1990    9        23            7            1721       1719          2201       220
    1990    9        27            4             645        645           802        81
    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

ans =

    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRS
    ____    _____    __________    _________    _______    __________    _______    ___

    1990    9        23            7            1721       1719          2201       220
    1990    9        27            4             645        645           802        81
    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
    1990    9        11            2             908        910          1613       155
    1990    9        22            6            1801       1750          2005       193
```

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 Data Using a `DatabaseDatastore` Object" on page 6-71
- "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 database, specified as a `DatabaseDatastore` object created using `databaseDatastore`.

More About

- DatabaseDatastore

See Also

`close` | `database` | `databaseDatastore` | `read`

Introduced in R2014b

isopen

Determine if database connection or database cursor is open

Syntax

```
o = isopen(conn)
o = isopen(curs)
```

Description

`o = isopen(conn)` determines if the database connection `conn` is open.

`o = isopen(curs)` determines if the database cursor `curs` is open.

Examples

Determine If the Database Connection Is Open

Create a database connection `conn` using the native ODBC interface. This code assumes that you are connecting to a data source named `dbtoolboxdemo` with user name `admin` and password `admin`.

```
conn = database.ODBCConnection('dbtoolboxdemo','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.

Close the database connection `conn`.

```
close(conn)
```

Determine if the database connection `conn` is closed.

```
o = isopen(conn)
```

```
o =

     0
```

`o` returns as the scalar `0` that denotes the database connection is closed. If the database connection is invalid, `o` returns the same result.

Determine If the Database Cursor Is Open

Create a database connection `conn` using the JDBC driver. This code assumes that you are connecting to a MySQL database named `test_db` with user name `username` and password `pwd`. Here, the machine name is `remotehost`. `test_db` contains the table `productTable` with the column `productDescription`.

```
conn = database('test_db','username','pwd','Vendor','MySQL',...
                'Server','remotehost');
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Create a database cursor object `curs` by running an SQL query in the database. Select data in `productDescription` from `productTable` using the database connection `conn`.

```
curs = exec(conn,'select productDescription from productTable');
```

Determine if the database cursor object `curs` is open.

```
o = isopen(curs)
```

```
o =

     1
```

`o` returns as the scalar `1` that denotes the database cursor is open.

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

```
close(curs)
```

Determine if the database cursor `curs` is closed.

```
o = isopen(curs)

o =

     0
```

o returns as the scalar 0 that denotes the database cursor is closed. If the database cursor is invalid, o returns the same result.

Close the database connection conn.

```
close(conn)
```

- “Import Data from Databases into MATLAB” on page 6-4
- “Export Data to New Record in Database” on page 6-22

Input Arguments

conn — Database connection
database connection object

Database connection, specified as a database connection object created using database.

curs — Database cursor
database cursor object

Database cursor, specified as an open SQL database cursor object created using exec.

Output Arguments

o — Open indicator
1 | 0

Open indicator, returned as the preceding scalar values. 1 means that the database connection or database cursor is open. 0 means that the database connection or database cursor is closed or invalid.

More About

- “Connecting to a Database” on page 2-191

- “Connecting to a Database Using the Native ODBC Interface” on page 3-18

See Also

`close` | `database` | `exec` | `isreadonly` | `ping`

Introduced in R2015b

preview

Return subset of data from `DatabaseDatastore`

Compatibility

`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.

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.

Examples

Preview Data

Using a JDBC driver, create a database connection `conn` to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to a database `toy_store`, a 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 `dbds` using the database connection `conn` and SQL query `sqlquery`. This SQL query reads all data from the table `airlinesmall`.

```
sqlquery = 'select * from airlinesmall';
```

```
dbds = databaseDatastore(conn,sqlquery);
```

Preview the first eight records in the data set returned by executing the SQL query `sqlquery` in the `DatabaseDatastore` object `dbds`.

```
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
```

Close the `DatabaseDatastore` object and database connection.

```
close(dbds)
```

- “Import Data Using a `DatabaseDatastore` Object” on page 6-71
- “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 database, specified as a `DatabaseDatastore` object created using `databaseDatastore`.

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.

More About

- DatabaseDatastore

See Also

`close` | `database` | `databaseDatastore` | `read`

Introduced in R2014b

read

Read data in `DatabaseDatastore`

Compatibility

The syntax `data = read(dbds,rowcount)` has been removed. Set the `DatabaseDatastore` property `ReadSize` instead.

`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.

Syntax

```
data = read(dbds)
[data,info] = read(dbds)
```

Description

`data = read(dbds)` returns data from the `DatabaseDatastore` object in increments specified using the property `ReadSize` of the `DatabaseDatastore` object. Subsequent calls to the `read` function continue reading from the endpoint of the previous call.

`[data,info] = read(dbds)` returns database information `info`.

Examples

Read Data

Using a JDBC driver, create a database connection `conn` to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to a database `toy_store`, a 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 `dbds` using the database connection `conn` and SQL query `sqlquery`. This SQL query retrieves all data from the table `airlinesmall`. 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 `dbds`.

```
data = read(dbds)
```

```
data = 

    Year     Month     DayofMonth     DayOfWeek     DepTime     CRSDepTime     ArrTime     CRS
    ____     _____     __________     _________     _______     __________     _______     ___

    1987     10        28             3             1140        1140           1212        121
    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        181
```

`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 `conn` to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to a database `toy_store`, a 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 `dbds` using the database connection `conn` and SQL query `sqlquery`. This SQL query retrieves all data from the table `airlinesmall`. 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 `dbds` and retrieve information `info` about the database.

```
[data,info] = read(dbds)


data = 

    Year     Month     DayofMonth     DayOfWeek     DepTime     CRSDepTime     ArrTime     CRS
    ____     _____     __________     _________     _______     __________     _______     ___

    1987     10        28             3             1140        1140           1212        121
    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        181


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 Data Using a `DatabaseDatastore` Object” on page 6-71
- “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 database, specified as a `DatabaseDatastore` object created using `databaseDatastore`.

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

More About

- DatabaseDatastore

See Also

`close` | `database` | `databaseDatastore` | `hasdata`

Introduced in R2014b

readall

Read all data in `DatabaseDatastore`

Compatibility

`readall` returns data as a table only. `readall` ignores database preference settings for data return formatting.

Syntax

```
data = readall(dbds)
```

Description

`data = readall(dbds)` returns all the data in the `DatabaseDatastore` object `dbds`, and resets the `DatabaseDatastore` to the point where no data has been read from it.

Examples

Read All Data in DatabaseDatastore Object

Using a JDBC driver, create a database connection `conn` to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to a database `toy_store`, a 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 `dbds` using the database connection `conn` and SQL query `sqlquery`. This SQL query reads all data from the table `airlinesmall`.

```
sqlquery = 'select * from airlinesmall';

dbds = databaseDatastore(conn,sqlquery);
```

Read all data in the `DatabaseDatastore` object `dbds`.

```
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    CRS
    ____    _____    __________    _________    _______    __________    _______    ___

    1987    10       28            3            1140       1140          1212       121
    1987    10        9            5            1155       1155          1250       130
    1987    10       22            4             715        715           807        80
```

Close the `DatabaseDatastore` object and database connection.

```
close(dbds)
```

- “Import Data Using a `DatabaseDatastore` Object” on page 6-71
- “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 database, specified as a `DatabaseDatastore` object created using `databaseDatastore`.

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.

More About

- DatabaseDatastore

See Also

`close` | `database` | `databaseDatastore` | `preview` | `read`

Introduced in R2014b

reset

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 `conn` to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to a database `toy_store`, a 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 `dbds` using the database connection `conn` and SQL query `sqlquery`. This SQL query retrieves all data from the table `airlinesmall`. 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       121
    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       181
```

`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 re-reading 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       121
    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       181
```

`read` again returns the first 10 records in the data set.

Close the `DatabaseDatastore` object and the database connection.

```
close(dbds)
```

- "Import Data Using a `DatabaseDatastore` Object" on page 6-71
- "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 database, specified as a `DatabaseDatastore` object created using `databaseDatastore`.

More About

- DatabaseDatastore

See Also

`close` | `database` | `databaseDatastore` | `exec` | `read`

Introduced in R2014b

DatabaseDatastore

Datastore for data in database

Compatibility

The `Cursor` property of the `DatabaseDatastore` object has been removed without replacement.

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 resulting data from executing an SQL query in a relational database. For details about the other datastores, see “Getting Started with Datastore”.

With a `DatabaseDatastore` object, you can:

- Preview data.
- Read data in chunks.
- Read every record in the data set.
- Reset the cursor position to the start of the data set.
- Continue reading data until the cursor position reaches the end of the data set.
- Analyze a large data set stored in a database using tall arrays or MapReduce.

Reading data from `DatabaseDatastore` objects is the same as executing `exec` and `fetch` on the data set. Here are the advantages of using `DatabaseDatastore` objects:

- Work with databases containing large amounts of data.
- Analyze large amounts of data using tall arrays with common MATLAB functions, such as `mean` and `histogram`. Create a tall array using `tall`. For details, see “Tall Arrays”.
- Write MapReduce algorithms that define the chunking and reduction of large amounts of data using `mapreduce`. For details, see “Getting Started with MapReduce”. For an example, see “Analyze Large Data in Database Using

MapReduce". For more MapReduce examples, see Building Effective Algorithms with MapReduce.

Create Object

Syntax

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

database connection object

Database connection, specified as a database connection object created using `database`.

`sqlquery` — SQL statement

character vector

SQL statement, specified as a character vector.

Data Types: `char`

`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`

Limitation

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”.

Properties

`Connection` — Database connection
connection object

Database connection, specified as a connection object created using `database`.

`Cursor` — Database cursor
database cursor object

Database cursor, specified as a database cursor object created using `exec` with the SQL query `query`.

Note: This property has been removed without replacement.

`Query` — SQL query
character vector

SQL query, specified as a character vector to denote the SQL query to execute in the database.

Data Types: `char`

`VariableNames` — Column names of the retrieved data table
cell array of character vectors

Column names of the retrieved data table, specified as a cell array of one or more character vectors.

Data Types: `char`

`ReadSize` — Number of rows to read
10000 (default) | numeric scalar

Number of rows to read from the retrieved data table, specified as a nonnegative numeric scalar. To specify the number of rows to read, set the `ReadSize` property.

Example: `dbds.ReadSize = 5000;`

Data Types: `double`

Object Functions

hasdata	Determine if data in DatabaseDatastore is available to read
preview	Return subset of data from DatabaseDatastore
read	Read data in DatabaseDatastore
readall	Read all data in DatabaseDatastore
reset	Reset DatabaseDatastore to initial state
close	Close database and driver resource utilizer

Examples

Create `DatabaseDatastore` Object

Using a JDBC driver, create a database connection `conn` to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to a database `toy_store`, a 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 `dbds` using the database connection `conn` and SQL query `sqlquery`. This SQL query retrieves all data from the table `airlinesmall`.

```
sqlquery = 'select * from airlinesmall';
```

```
dbds = databaseDatastore(conn,sqlquery)
```

```
dbds = 

  DatabaseDatastore with properties:

       Connection: [1×1 database.jdbc.connection]
            Query: 'select * from airlinesmall'
    VariableNames: {1×29 cell}
         ReadSize: 10000
```

`databaseDatastore` executes the SQL query. `dbds` contains these properties:

- Database connection object
- Executed SQL query
- List of column names from the executed SQL query
- Maximum number of records to read from the executed SQL query

Display the database connection property `Connection`.

```
dbds.Connection
```

```
ans = 

  connection with properties:

       Instance: 'toy_store'
       UserName: ''
         Driver: []
            URL: []
    Constructor: [1×1 com.mathworks.toolbox.database.databaseConnect]
        Message: []
         Handle: [1×1 com.microsoft.sqlserver.jdbc.SQLServerConnection]
        TimeOut: 0
     AutoCommit: 'on'
           Type: 'Database Object'
```

The `Message` property is blank when the database connection is successful.

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 `conn` to a Microsoft® SQL Server® database with Windows® authentication. Specify a blank user name and password. The code assumes that you are connecting to a database `toy_store`, a 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 `dbds` using the database connection `conn` and SQL query `sqlquery`. This SQL query retrieves all data from the table `airlinesmall`. Specify reading a maximum of 1000 records from the executed SQL query when using the `read` function.

```
sqlquery = 'select * from airlinesmall';

dbds = databaseDatastore(conn,sqlquery,'ReadSize',1000)
```

```
dbds = 

  DatabaseDatastore with properties:

       Connection: [1×1 database.jdbc.connection]
            Query: 'select * from airlinesmall'
    VariableNames: {1×29 cell}
         ReadSize: 1000
```

`databaseDatastore` executes the SQL query. `dbds` contains these properties:

- Database connection object
- Executed SQL query
- List of column names from the executed SQL query
- Maximum number of records to read from the executed SQL query

Display the database connection property `Connection`.

```
dbds.Connection
```

```
ans = 

  connection with properties:

       Instance: 'toy_store'
       UserName: ''
         Driver: []
            URL: []
    Constructor: [1×1 com.mathworks.toolbox.database.databaseConnect]
        Message: []
         Handle: [1×1 com.microsoft.sqlserver.jdbc.SQLServerConnection]
        TimeOut: 0
     AutoCommit: 'on'
           Type: 'Database Object'
```

The `Message` property is blank when the database connection is successful.

Close the `DatabaseDatastore` object and the database connection.

```
close(dbds)
```

- “Import Data Using a `DatabaseDatastore` Object” on page 6-71
- “Analyze Large Data in Database Using Tall Arrays”
- “Analyze Large Data in Database Using MapReduce”

See Also

`database` | `exec`

More About

- “Getting Started with Datastore”
- “Getting Started with MapReduce”

Introduced in R2014b

datainsert

Export MATLAB data into database table

To export MATLAB data into a database, use these functions: `datainsert`, `fastinsert`, and `insert`. For maximum performance, use `datainsert`.

For other differences among these functions, see “Inserting Data Using the Command Line” on page 2-197.

Syntax

```
datainsert(conn,tablename,colnames,data)
```

Description

`datainsert(conn,tablename,colnames,data)` inserts data from the MATLAB workspace into a database table.

Examples

Export MATLAB Cell Array Data

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 user name `username` and password `pwd`. This database contains the table `inventoryTable` with these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

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

Display the last rows in `inventoryTable` before insertion of 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 cell array of column names for the database table `inventoryTable`.

```
colnames = {'productNumber','Quantity','Price','inventoryDate'};
```

Define the 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 database connection `conn`.

```
tablename = 'inventoryTable';

datainsert(conn,tablename,colnames,data)
```

Display 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 connection.

```
close(conn)
```

Export MATLAB Table Data

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`. 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 insertion of 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 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 database connection `conn`.

```
tablename = 'inventoryTable';
```

```
datainsert(conn,tablename,colnames,data)
```

Display 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 connection.

```
close(conn)
```

Export MATLAB Structure Data

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 user name `username` and password `pwd`. This database contains the table `inventoryTable` with these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

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

Display the last rows in `inventoryTable` before insertion of 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 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 database connection `conn`.

```
tablename = 'inventoryTable';

datainsert(conn,tablename,colnames,data)
```

Display 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 connection.

```
close(conn)
```

Export MATLAB Numeric Matrix Data

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`. 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 database connection `conn`.

```
tablename = 'salesVolume';

datainsert(conn,tablename,colnames,data)
```

Display 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 connection.

```
close(conn)
```

- "Export Data to New Record in Database" on page 6-22
- "Export Multiple Records from the MATLAB Workspace" on page 6-27

- "Export Data Using Bulk Insert" on page 6-31
- "Replace Existing Data in a Database" on page 6-25
- "Roll Back Data After Updating a Record" on page 6-19

Input Arguments

`conn` — Database connection
database connection object

Database connection, specified as a database connection object created using `database`.

`tablename` — Database table name
character vector

Database table name, specified as a character vector denoting the name of a table in your database.

Data Types: `char`

`colnames` — Database table column names
cell array of character vectors

Database table column names, specified as a cell array of one or more character vectors to denote the columns in the existing database table `tablename`.

Example: `{'col1','col2','col3'}`

Data Types: `cell`

`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 or the JDBC/ODBC bridge, 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 `NaN`s, convert these entries to an empty value `''`.

MATLAB date numbers and `NaN`s are supported for insert when `data` is a numeric matrix. Date numbers inserted into database date and time columns convert to `java.sql.Date`. Any converted date and time data is accurately converted back to the native database format in the target database upon insertion.

If `data` is a structure, field names in the structure must match `colnames`. If `data` is a table or a dataset array, the variable names in the table or dataset array must match `colnames`.

More About

Tips

- When you establish a database connection using a JDBC driver or the JDBC/ODBC bridge, `datainsert` performs faster than `fastinsert`.
- To insert dates and timestamps with the native ODBC interface, use the format `'YYYY-MM-DD HH:MM:SS.MS'`.
- `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 status of the `AutoCommit` flag determines whether `datainsert` commits the data to the database. View the status of this flag using `get`. To change the status of this flag, use `set`. To commit the insert data to the database, use `commit` or run an SQL COMMIT statement using `exec`. To roll back the insert data from the database, use `rollback` or run an SQL ROLLBACK statement using `exec`.

- "Inserting Data Using the Command Line" on page 2-197

See Also

`close` | `database` | `fastinsert` | `insert` | `update`

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 from the database to the MATLAB workspace. 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 JDBC and ODBC data sources.
- Establish multiple connections to databases.
- Select tables and columns of interest.
- Fine-tune selection using SQL query criteria.
- Preview selected data.
- Import selected data into MATLAB workspace.
- Save generated SQL queries.
- Generate MATLAB code.

To use Database Explorer for the first time, migrate from Visual Query Builder and set Database Explorer preferences to initialize the app. For details, see “Working with Database Explorer” on page 4-2.

Open the Database Explorer App

- MATLAB Toolstrip: On the **Apps** tab, under **Database Connectivity and Reporting**, click the app icon.
- MATLAB command prompt: Enter `dexplore`.

Examples

Display Data from a Single Database Table

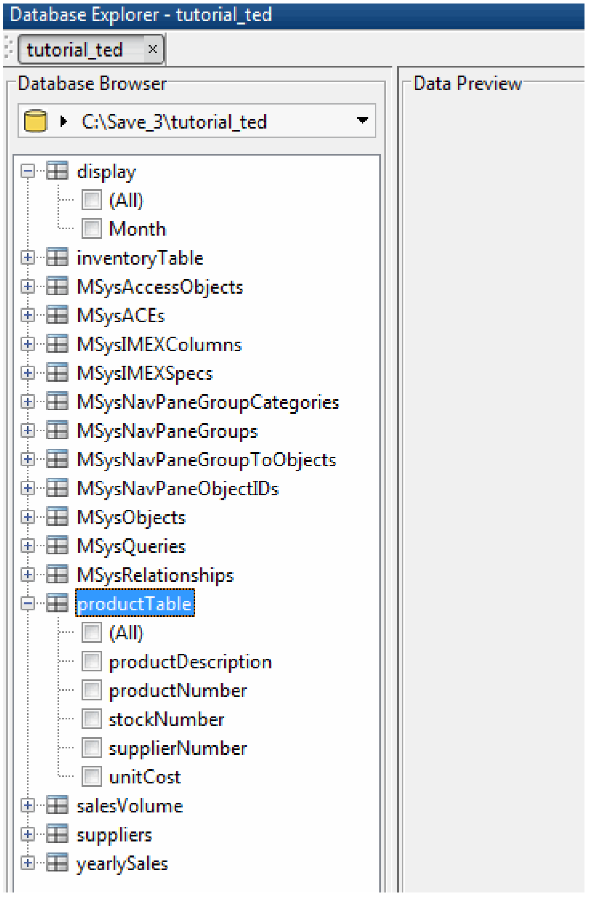
Set up the data source for the `tutorial.mdb` database and connect to this database. For details, see "Microsoft Access ODBC for Windows" on page 2-19.

Display data in the **Data Preview** pane by opening the database table of interest in the **Database Browser** pane on the Database Explorer Toolstrip. When you select a database table in the **Database Browser** pane, the table is highlighted and a corresponding entry displays in the **SQL Criteria** panel. Enter query conditions for the selected table in the **SQL Criteria** panel.

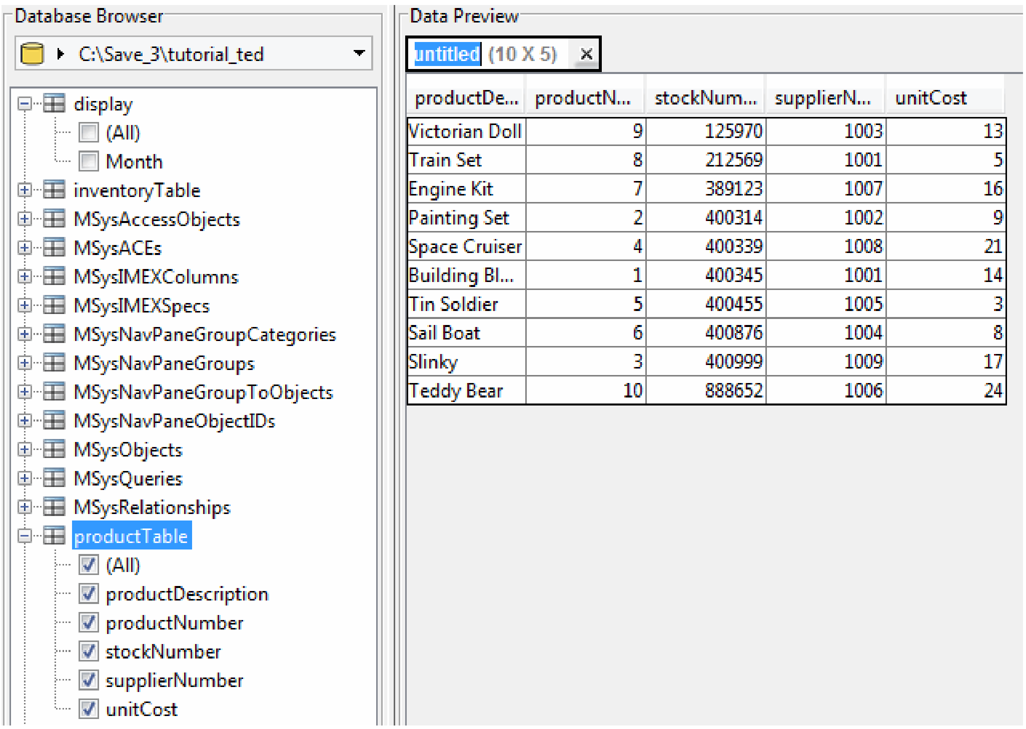
For any given table, you can select the table information in these ways:

- Click to highlight the database table name. Clicking the database table name does not display data in the **Data Preview** pane but does update the **SQL Criteria** panel.
- Select **(All)** to choose all table columns and display them in the **Data Preview** pane.
- Select specific check boxes to choose individual table columns and display them in the **Data Preview** pane.

Note: The order of the columns in the **Data Preview** pane matches the order in which you select them in the **Database Browser** pane.



Select **(All)** to choose all database columns or select check boxes for specific table columns.



productDe...	productN...	stockNum...	supplierN...	unitCost
Victorian Doll	9	125970	1003	13
Train Set	8	212569	1001	5
Engine Kit	7	389123	1007	16
Painting Set	2	400314	1002	9
Space Cruiser	4	400339	1008	21
Building Bl...	1	400345	1001	14
Tin Soldier	5	400455	1005	3
Sail Boat	6	400876	1004	8
Slinky	3	400999	1009	17
Teddy Bear	10	888652	1006	24

To change your display, select or clear check boxes in the **Database Browser** pane. The data updates in the **Data Preview** pane.

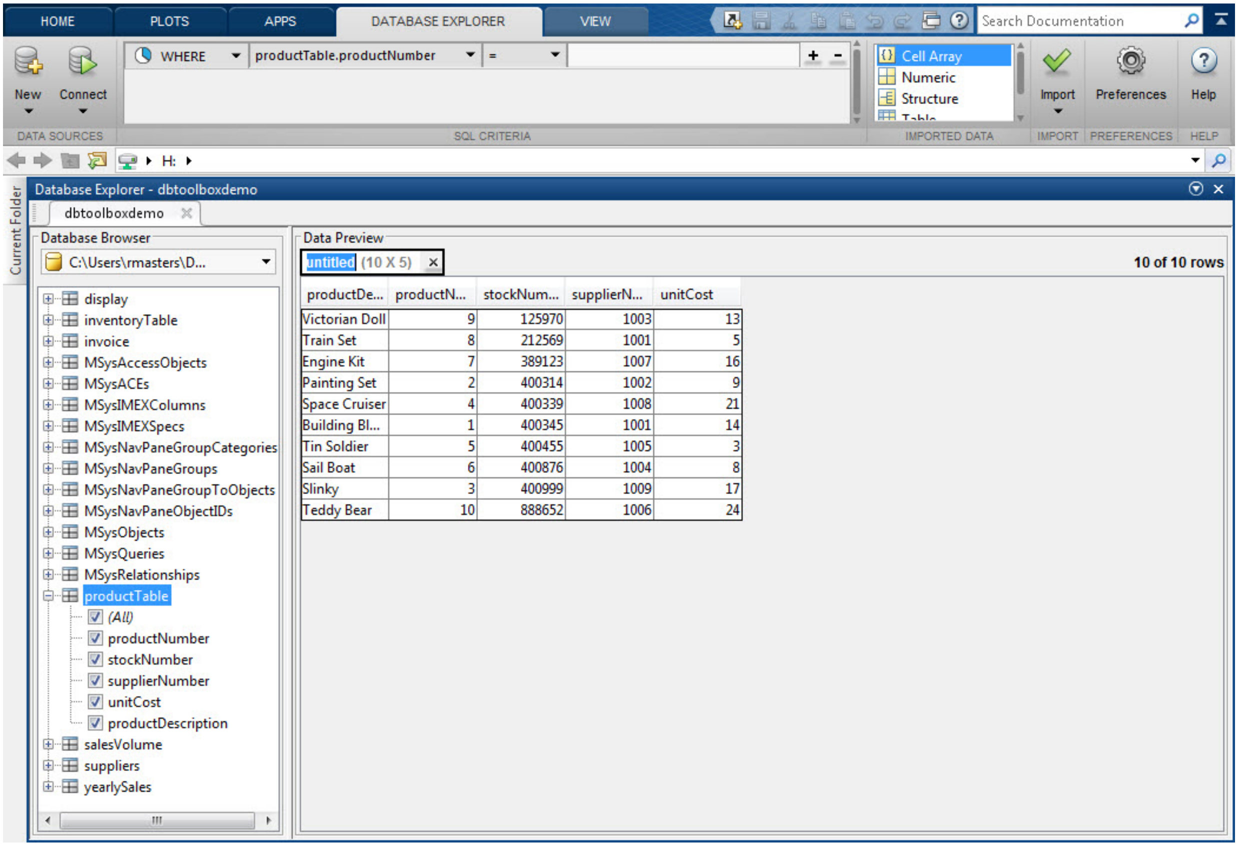
The **Data Preview** pane displays a limited number of rows. The total number of rows selected in the database appears at the right of the display. You can change the display size by clicking **Preferences** and adjusting the **Data Preview size**.

Close the database connection. For details, see “Configuring a Driver and Data Source” on page 2-16.

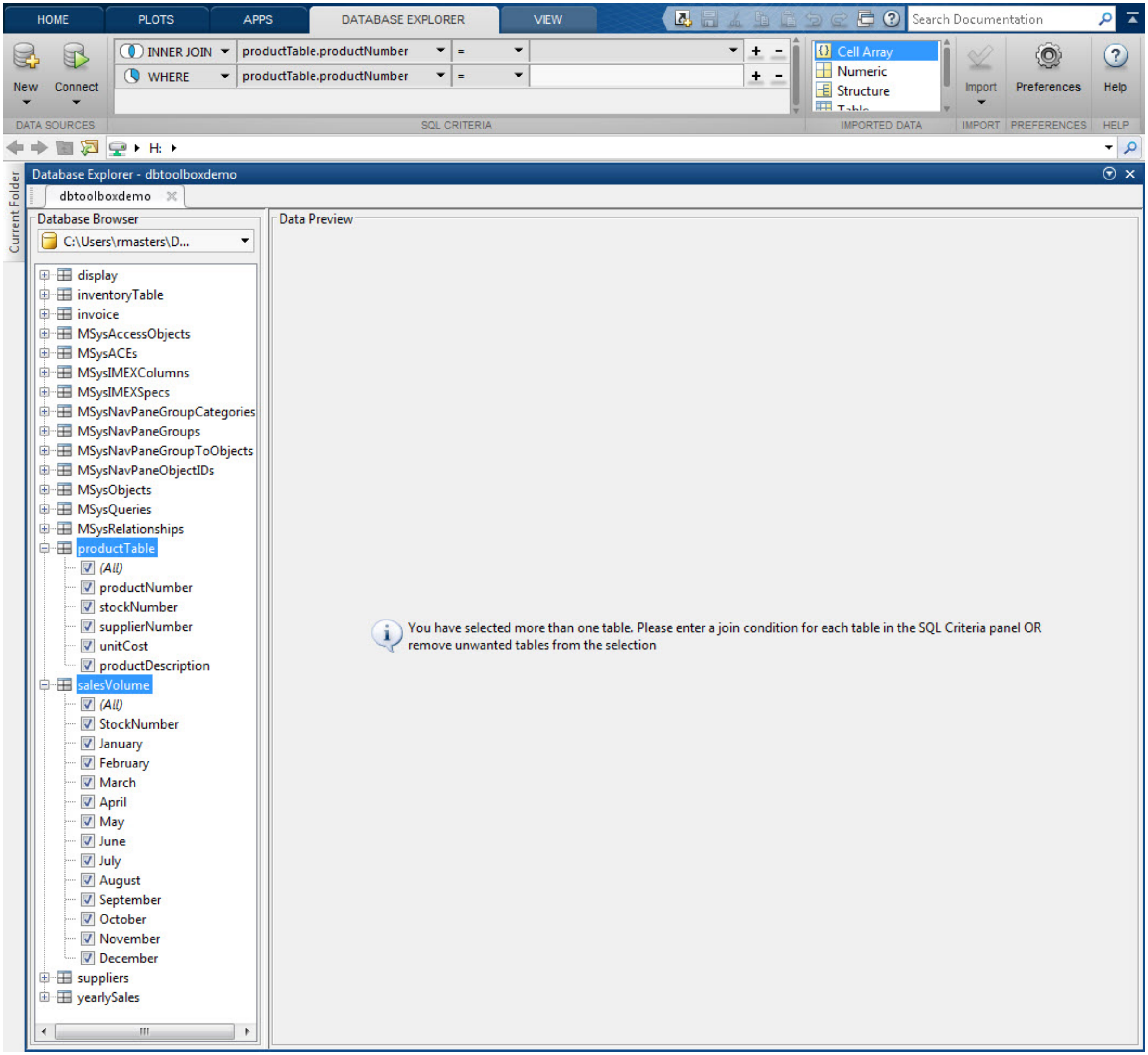
Join Data from Multiple Database Tables

Set up the data source for the `tutorial.mdb` database and connect to this database. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

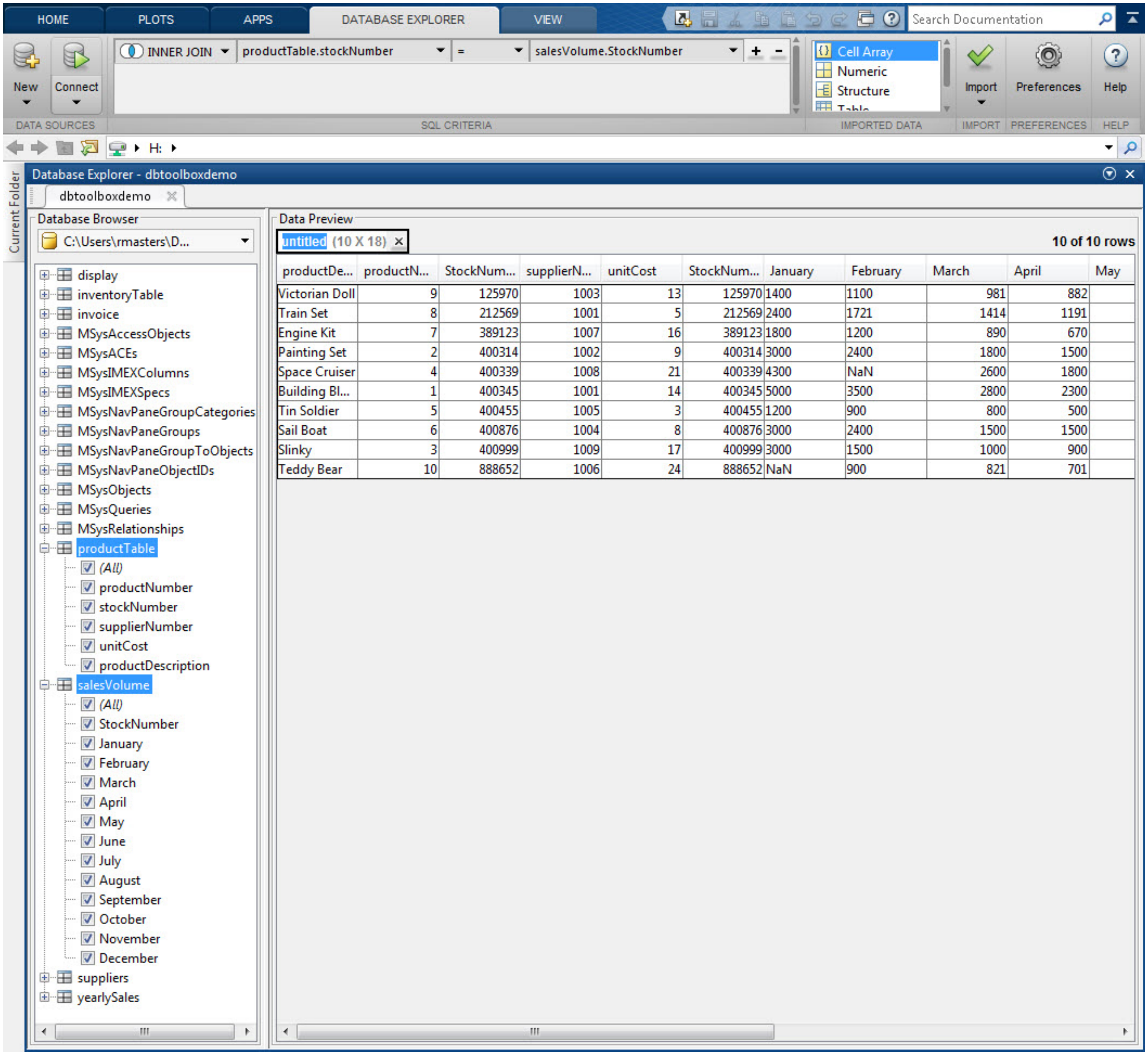
Display data in the **Data Preview** pane by opening the desired database table in the **Database Browser** pane. The **SQL Criteria** panel updates in the Database Explorer Toolstrip.



When you select additional tables in the **Database Browser** pane, the **SQL Criteria** panel updates.



Display the contents for the selected tables using the **SQL Criteria** panel to define a join of the selected tables. Click the drop-down lists to specify the table column for joining the selected tables. The join results appear in the **Data Preview** pane.

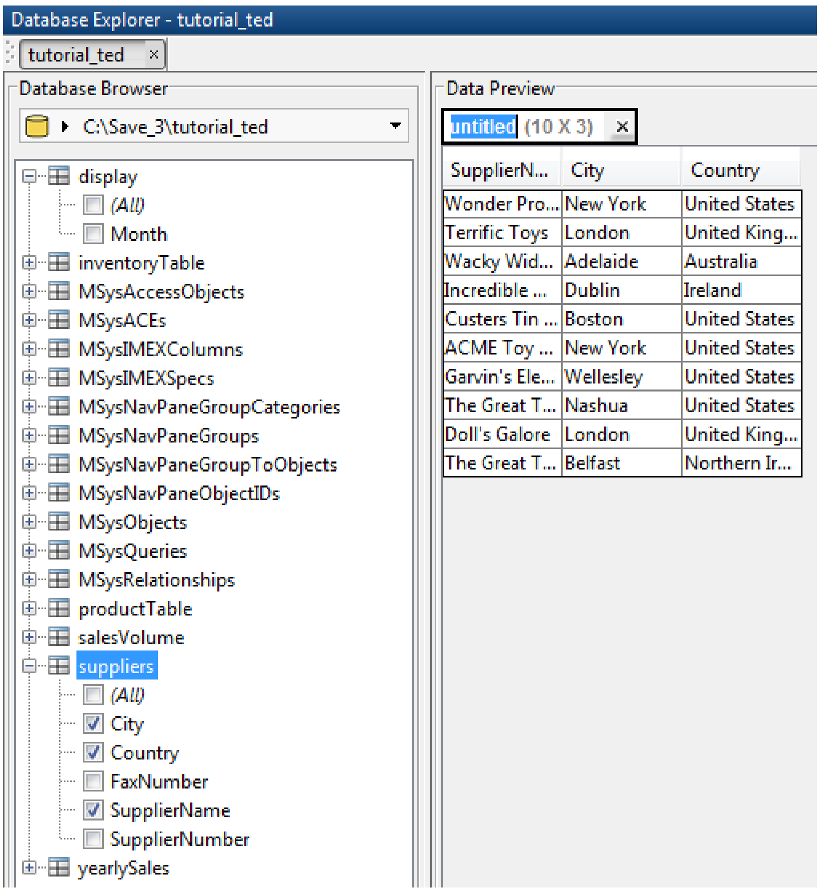

HOME
PLOTS
APPS
DATABASE EXPLORER
VIEW
Search Documentation
New
Connect
DATA SOURCES
INNER JOIN
productTable.stockNumber
=
salesVolume.StockNumber
SQL CRITERIA
Cell Array
Numeric
Structure
IMPORTED DATA
Import
IMPORT
Preferences
PREFERENCES
Help
HELP
H:
Current Folder
Database Explorer - dbtoolboxdemo
dbtoolboxdemo
Database Browser
C:\Users\rmasters\D...
display
inventoryTable
invoice
MSysAccessObjects
MSysACEs
MSysIMEXColumns
MSysIMEXSpecs
MSysNavPaneGroupCategories
MSysNavPaneGroups
MSysNavPaneGroupToObjects
MSysNavPaneObjectIDs
MSysObjects
MSysQueries
MSysRelationships
productTable
(All)
productNumber
stockNumber
supplierNumber
unitCost
productDescription
salesVolume
(All)
StockNumber
January
February
March
April
May
June
July
August
September
October
November
December
suppliers
yearlySales
Data Preview
untitled (10 X 18)
10 of 10 rows
productDe... productN... StockNum... supplierN... unitCost StockNum... January February March April May
Victorian Doll 9 125970 1003 13 125970 1400 1100 981 882
Train Set 8 212569 1001 5 212569 2400 1721 1414 1191
Engine Kit 7 389123 1007 16 389123 1800 1200 890 670
Painting Set 2 400314 1002 9 400314 3000 2400 1800 1500
Space Cruiser 4 400339 1008 21 400339 4300 NaN 2600 1800
Building Bl... 1 400345 1001 14 400345 5000 3500 2800 2300
Tin Soldier 5 400455 1005 3 400455 1200 900 800 500
Sail Boat 6 400876 1004 8 400876 3000 2400 1500 1500
Slinky 3 400999 1009 17 400999 3000 1500 1000 900
Teddy Bear 10 888652 1006 24 888652 NaN 900 821 701

Close the database connection. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

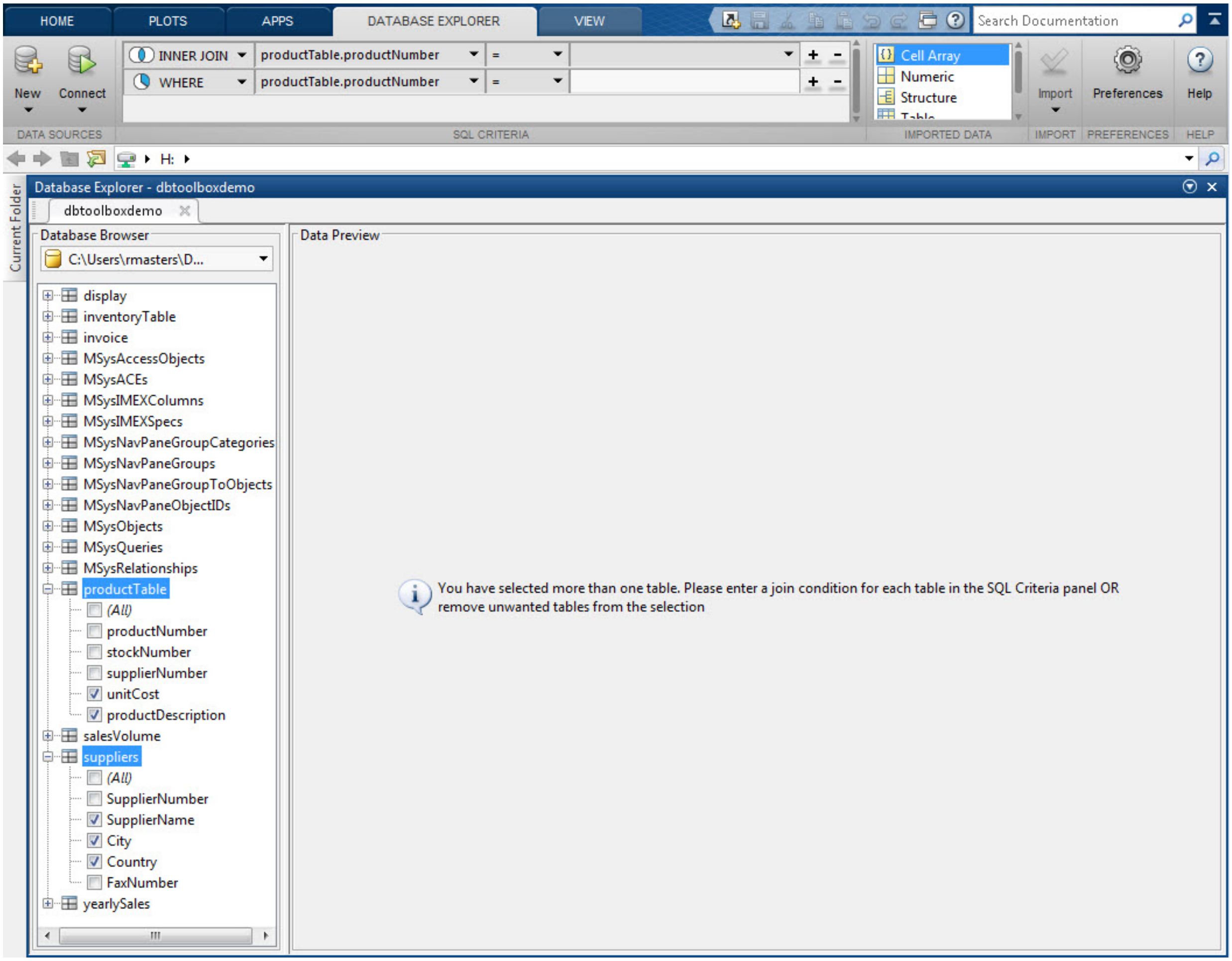
Query Data Using a Left Outer Join

Set up the data source for the `tutorial.mdb` database and connect to this database. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

Expand the table `suppliers` and select the fields `SupplierName`, `City`, and `Country`.

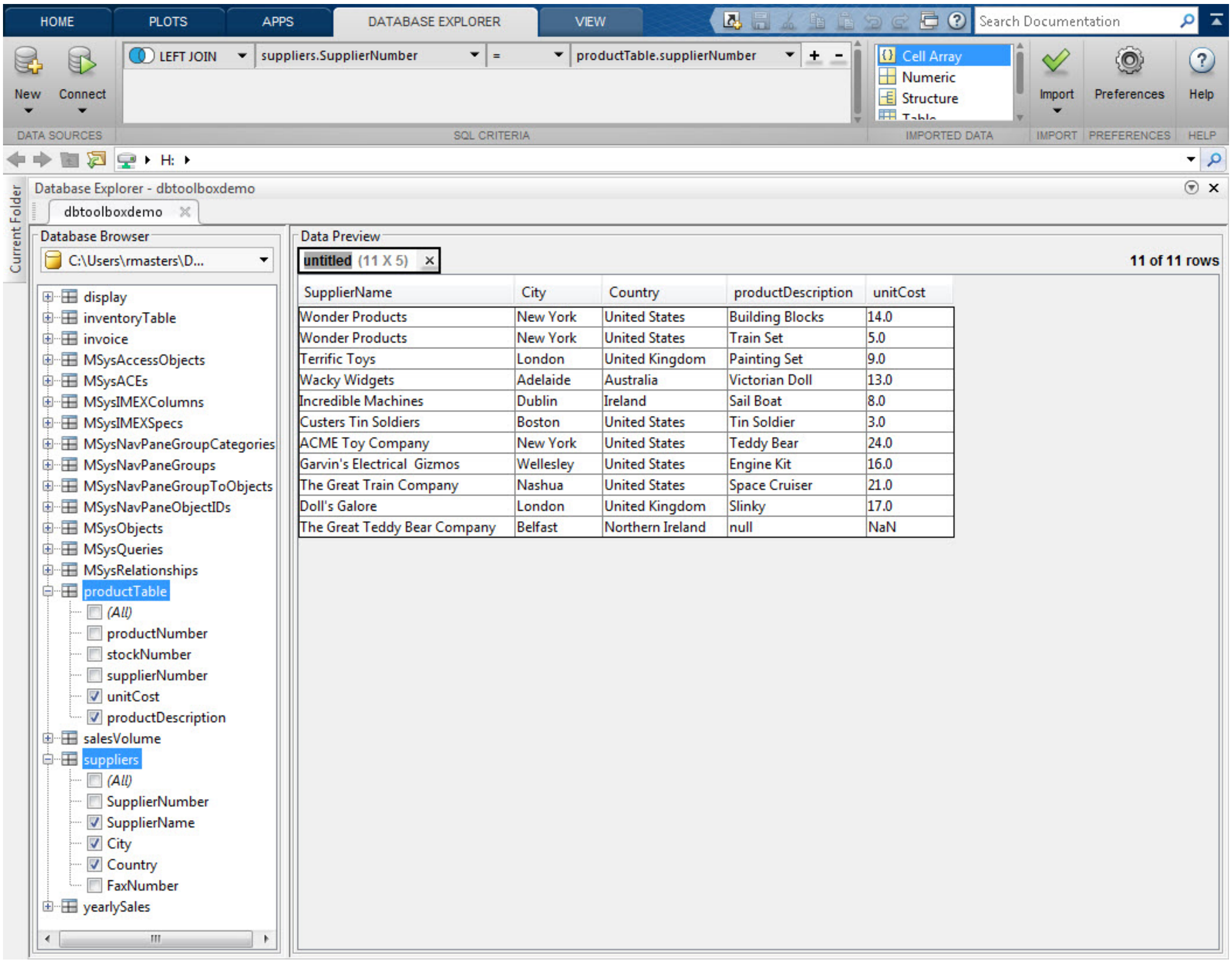


Expand the table `producttable` and select the fields `productDescription` and `unitCost`. The **Data Preview** pane displays a message prompting you to enter a join condition. There are two empty conditions in the **SQL Criteria** panel on the Database Explorer Toolstrip.

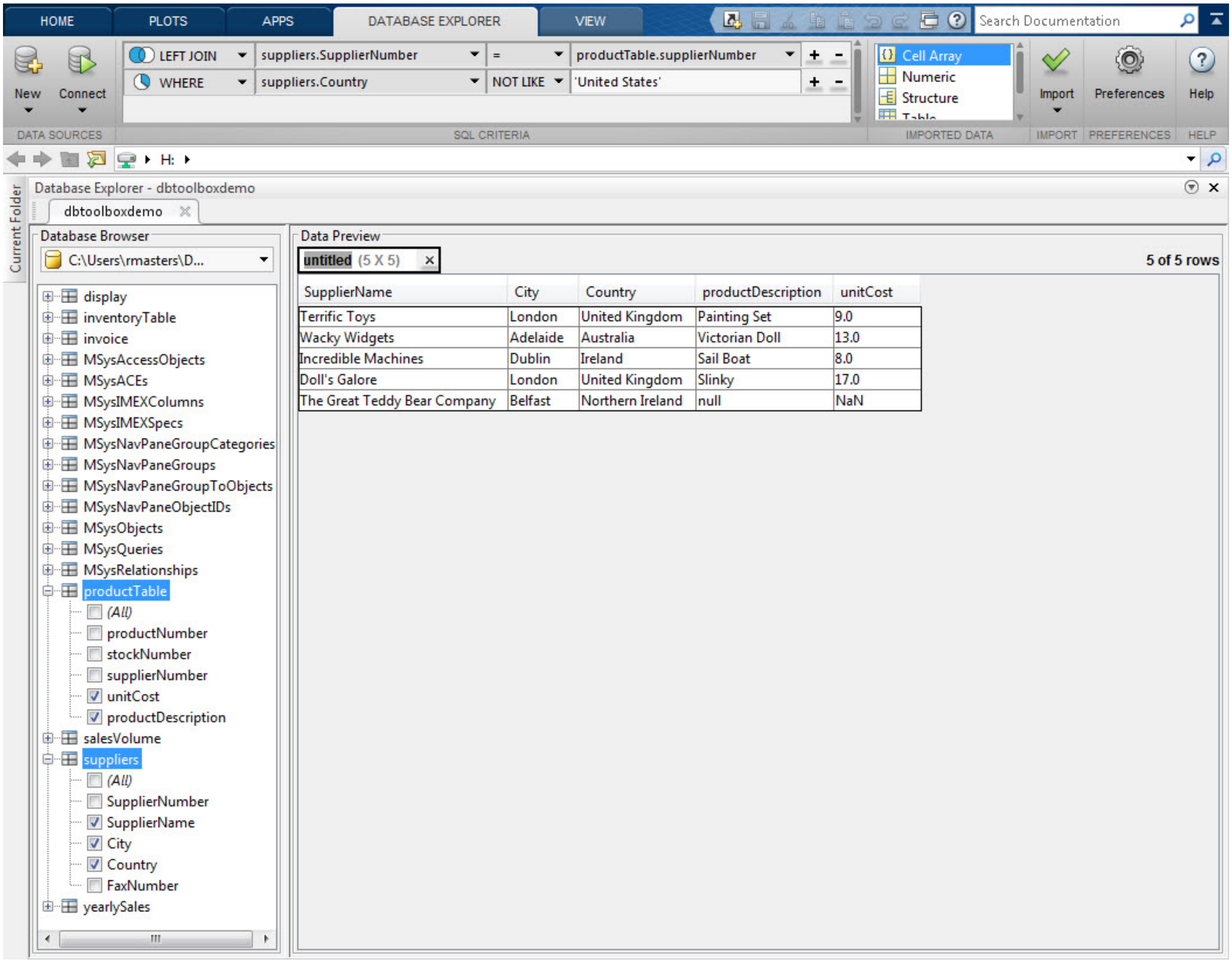


From the **SQL Criteria** panel, in the first condition at the top, change the first combo box for condition type to `LEFT JOIN`. Change the second combo box to `suppliers.SupplierNumber`. Change the last combo box to `producttable.SupplierNumber`. A left join, with the `suppliers` table on the left, implies that all the rows in the `suppliers` table are included in the final result. The rows in `suppliers` that do not have a match with any row in `producttable` are padded with null values in the final result.

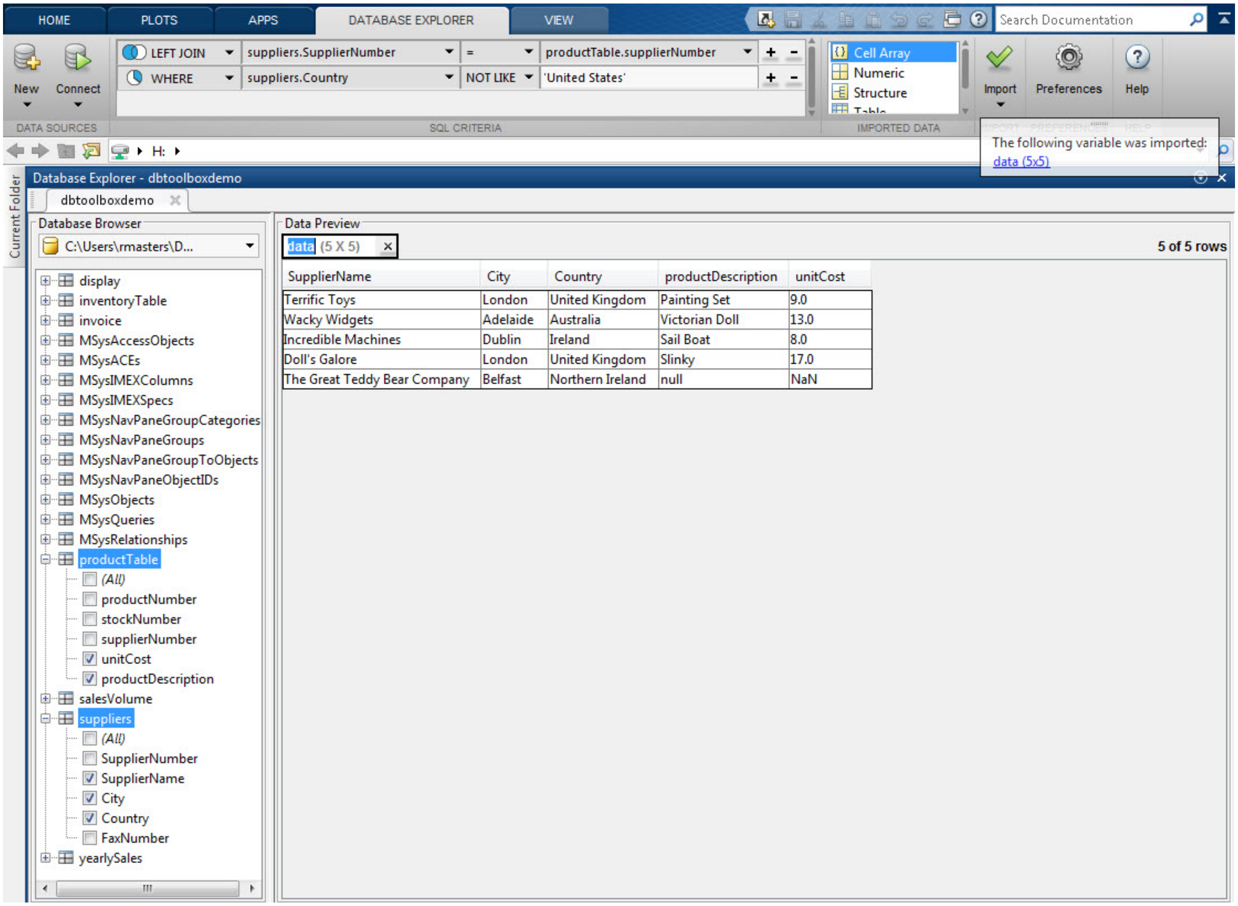
In the **Data Preview**, there are 11 rows that match the query conditions. There is a `null` in `productDescription` and a `NaN` in `unitCost` because the supplier `The Great Teddy Bear Company` supplies no products. If the condition type is set to `INNER JOIN` instead of `LEFT JOIN`, this row does not appear in the final result.



From the **SQL Criteria** pane, click **+** at the end of the `LEFT JOIN` condition to add a query condition. Change the first combo box to **WHERE**, the second to `suppliers.Country`, and the third to `NOT LIKE`. In the last text box, enter `United States` and then enter the new condition using the **Enter** or **Tab** keys. The query results appear in the **Data Preview** pane.



Enter the variable name as `data` in the text box `untitled` located above the table preview. Select **Import** > **Import** to import the data displayed in the **Data Preview** pane into MATLAB as a variable named `data`. For details about using the MATLAB Variables editor, see “Create and Edit Variables”.



Close the database connection. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

Import Data to the MATLAB Workspace

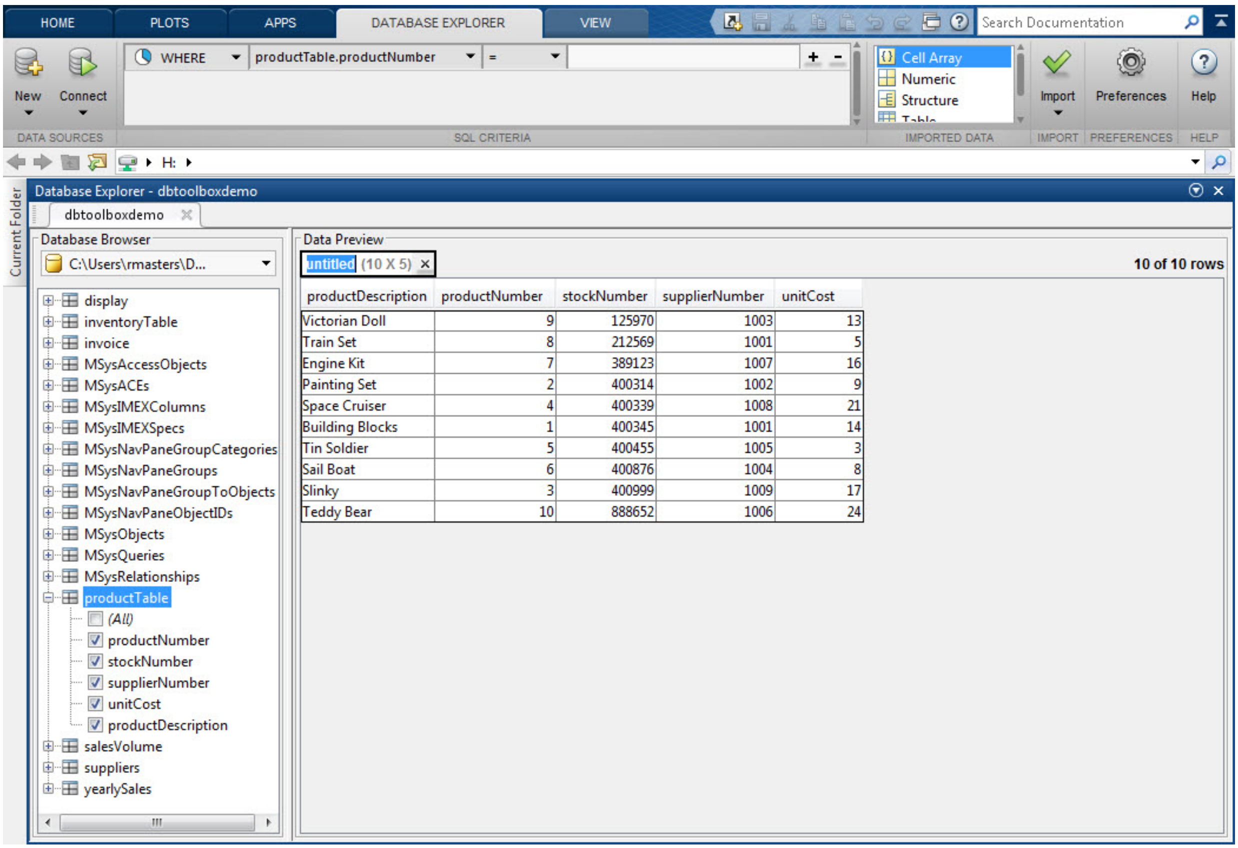
Set up the data source for the `tutorial.mdb` database and connect to this database. For details, see “Microsoft Access ODBC for Windows” on page 2-19.

Select data using the **Database Browser** pane from a single table. Or, create a query using the **SQL Criteria** panel. Display the results in the **Data Preview** pane.

Name the MATLAB variable by entering it in the **untitled** text box in the **Data Preview** pane.

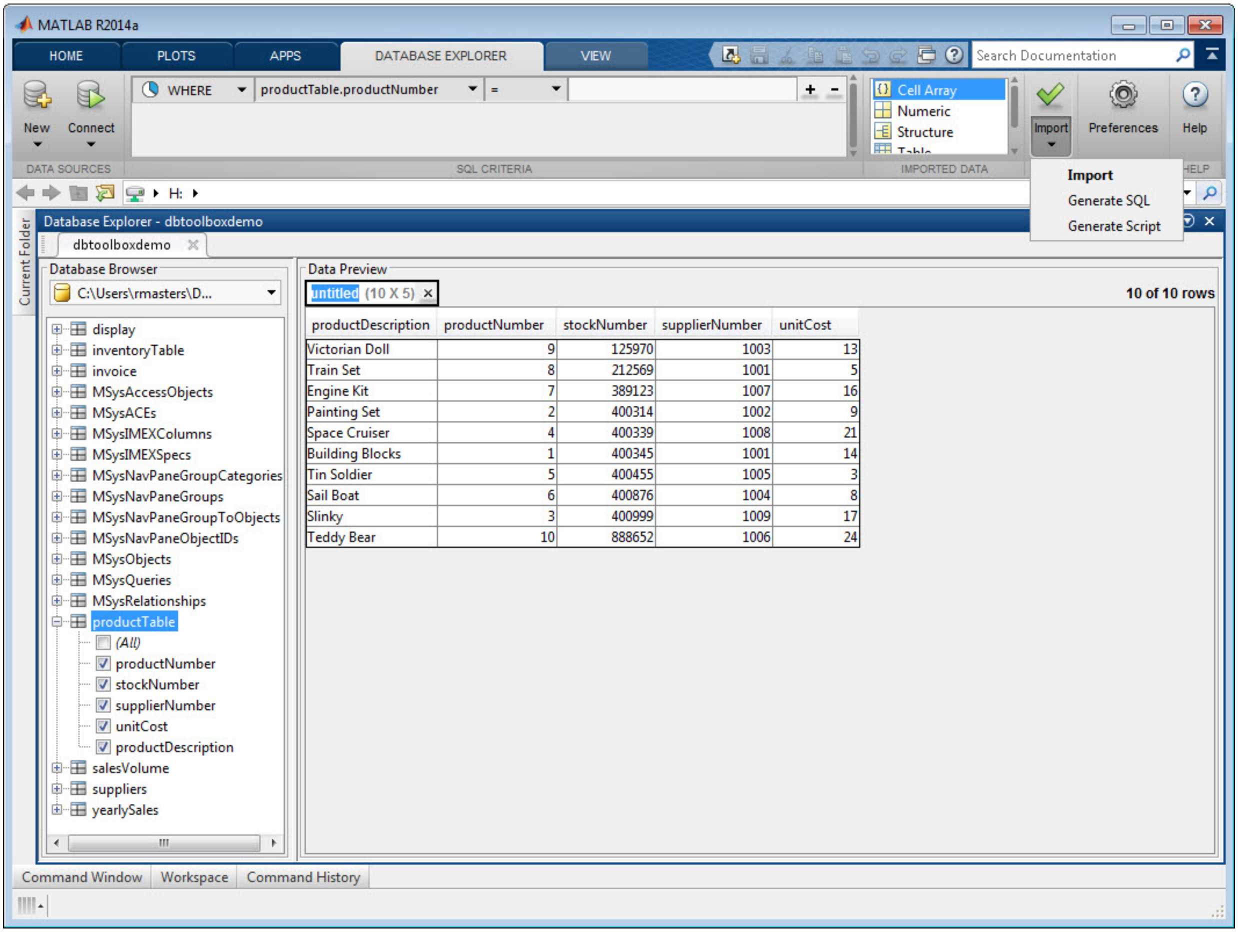
Define the data type for a MATLAB variable in the **Imported Data** panel to store the data displayed in the **Data Preview** pane. Supported data types are:

- **Cell Array**
- **Numeric**
- **Structure**
- **Table**
- **Dataset** (requires Statistics and Machine Learning Toolbox)

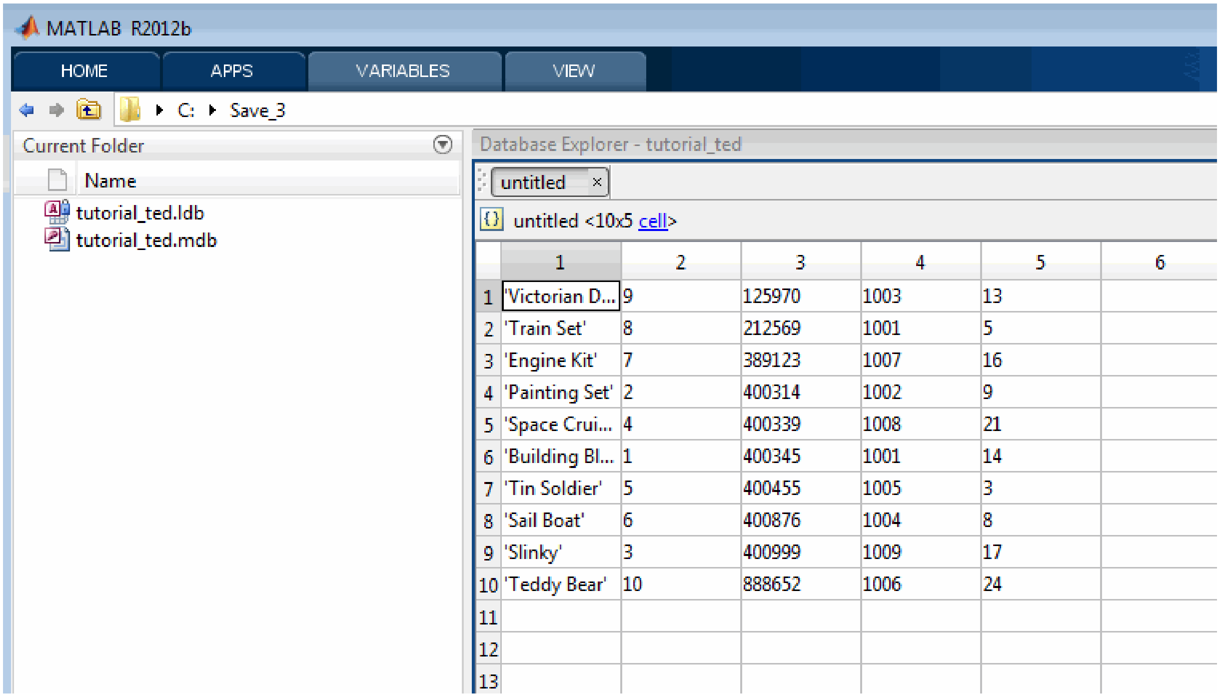


Select **Import** > **Import** to import the data displayed in the **Data Preview** pane.

Note: When importing large amounts of data, Database Explorer imports data in batches. The batch size is set to 1,000 rows by default. To change the batch size, click **Preferences** and adjust **Import batch size**.



Optionally, display the imported data in the MATLAB workspace using the Variables editor. For details about using the Variables editor, see “Create and Edit Variables”.



Optionally, manipulate the data using MATLAB functions.

Close the database connection. For details, see "Microsoft Access ODBC for Windows" on page 2-19.

More About

- "Connection Options" on page 2-9
- "Configuring a Driver and Data Source" on page 2-16

See Also

Functions
`close` | `database` | `exec` | `fetch`

Introduced in R2012b

dmd

Construct database metadata object

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.

For a list of functions that operate on database metadata objects, enter:

```
help dmd/Contents
```

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` | `supports` | `tables`

Introduced before R2006a

driver

Construct database driver object

Compatibility

`driver` has been removed.

Syntax

```
d = driver('s')
```

Description

`d = driver('s')` constructs a database driver object `d` from `s`, where `s` is a database URL string of the form `jdbc:odbc:`*`name`* or *`name`*. The driver object `d` is the first driver that recognizes `s`.

Examples

`d = driver('jdbc:odbc:thin:@144.212.123.24:1822:')` creates driver object `d`.

See Also

`get`

Introduced before R2006a

drivermanager

Construct database drivermanager object

Compatibility

`drivermanager` has been removed.

Syntax

`dm = drivermanager`

Description

`dm = drivermanager` constructs a database drivermanager object which comprises the properties for all loaded database drivers. Use `get` and `set` to obtain and change the properties of `dm`.

Examples

Create a database drivermanager object and return its properties.

```
dm = drivermanager
get(dm)
```

See Also

`get` | `set`

Introduced before R2006a

exec

Execute SQL statement and open cursor

`exec` executes SQL statements using the command line. To execute SQL statements interactively, use the Database Explorer app.

Syntax

```
exec(conn,sqlquery)
```

```
curs = exec(conn,sqlquery)
curs = exec(conn,sqlquery,qTimeOut)
curs = exec(conn,sqlquery,Name,Value)
```

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.

`curs = exec(conn,sqlquery)` executes the SQL statement `sqlquery` for the database connection `conn` and returns the cursor object `curs`.

`curs = exec(conn,sqlquery,qTimeOut)` executes the SQL statement with a timeout value `qTimeOut`.

`curs = exec(conn,sqlquery,Name,Value)` executes the SQL statement and creates a scrollable cursor.

Examples

Select Data Using the Native ODBC Interface

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 `dbtoolboxdemo` with `admin` as the user name and password.

```
conn = database.ODBCConnection('dbtoolboxdemo','admin','admin');
```

Select data from productTable that you access using the database.ODBCConnection object, conn. Assign the SQL statement to the variable sqlquery. Assign the returned cursor object to the variable curs.

```
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, exec returns curs as a database.ODBCCursor object instead of a Database Cursor Object.

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

```
close(curs)
```

Select Data Using a Scrollable Cursor

Using a MySQL database, select data from a table that you access using the native ODBC database connection conn and create a scrollable cursor.

Connect to the MySQL database. This code assumes that you are connecting to a data source named MySQL with user name username and password pwd.

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

Select all rows from the productTable table and create a scrollable cursor. Assign the SQL statement to the variable sqlquery. Assign the returned cursor object to the variable curs.

```
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 `exec` creates a scrollable cursor, display the hidden `Scrollable` property.

```
curs.Scrollable
```

```
ans =

     1
```

The `Scrollable` property equals 1 when the cursor is scrollable.

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

```
close(curs)
```

Select Data with a Timeout Value

Using the `dbtoolboxdemo` data source, select data from `productTable` that you access using the database connection `conn` with a timeout of 10 seconds. The timeout value specifies the maximum amount of time `exec` tries to execute the SQL statement. Assign the SQL statement to the variable `sqlquery`. Assign the returned cursor object to the variable `curs`.

```
sqlquery = 'select * from productTable';
curs = exec(conn,sqlquery,10)
```

```
curs =

  cursor with properties:

        Attributes: []
              Data: 0
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: 'select * from productTable'
           Message: []
              Type: 'Database Cursor Object'
         ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
```

```
      Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
   Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
       Fetch: 0
```

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

```
close(curs)
```

Use a Variable in a Query

Using the `dbtoolboxdemo` data source, select data from the `productTable` table that you access using the database connection `conn`, where `productdesc` is a variable. In this example, you are prompted to specify the product description. Your input is assigned to the variable `productdesc`.

```
productdesc = input('Enter your product description: ', 's')
```

The following prompt appears.

```
Enter your product description:
```

Type the following into the Command Window.

```
Train Set
```

To perform the query using your input, run the following code. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
sqlquery = ['select * from productTable'...
'where productDescription = ' '''' productdesc ''''];
curs = exec(conn,sqlquery);
curs = fetch(curs);
curs.Data
```

```
ans =

    [8]    [212569]    [1001]    [5]    'Train Set'
```

The select statement is created by using square brackets to concatenate the two character vectors `select * from productTable where productDescription =` and `'productdesc'`. To create the pair of single quotation marks that appears in the SQL 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.

Perform the query without a variable.

```
sqlquery = ['select * from productTable'...
'where productDescription = ' '''Engine Kit'''];
curs = exec(conn,sqlquery);
curs = fetch(curs);
curs.Data
```

```
ans =

    [7]    [389123]    [1007]    [16]    'Engine Kit'
```

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

```
close(curs)
```

Limit Maximum Number of Rows to Return

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 user name `username` and password `pwd`.

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

Select data from the table `productTable`. To limit the maximum number of rows, set the name-value pair argument `'maxRows'` to `10`.

```
sqlquery = 'select * from productTable order by productNumber';

curs = exec(conn,sqlquery,'maxRows',10);
```

`exec` retrieves only 10 rows from the database server.

Display the returned data `curs.Data`.

```
curs = fetch(curs);

curs.Data
```

```
ans =

    [ 1]    [400345]    [1001]    [14]    'Building Blocks'
    [ 2]    [400314]    [1002]    [ 9]    'Painting Set'
    [ 3]    [400999]    [1009]    [17]    'Slinky'
    [ 4]    [400339]    [1008]    [21]    'Space Cruiser'
    [ 5]    [400455]    [1005]    [ 3]    'Tin Soldier'
    [ 6]    [400876]    [1004]    [ 8]    'Sail Boat'
```

```
    [ 7]    [389123]    [1007]    [16]    'Engine Kit'
    [ 8]    [212569]    [1001]    [ 5]    'Train Set'
    [ 9]    [125970]    [1003]    [13]    'Victorian Doll'
    [10]    [888652]    [1006]    [24]    'Teddy Bear'
```

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

```
close(curs)
```

Call a 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 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 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 user name `username` and password `pwd`.

```
conn = database.ODBCConnection('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)
```

Create a Table Using the MATLAB® Interface to SQLite

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`.

Close the SQLite connection.

```
close(conn)
```


- “Import Data from Databases into MATLAB” on page 6-4
- “Create Queries with Characters and Variables” on page 6-8
- “Import Large Data Using Paging” on page 6-69
- “Call a Stored Procedure That Returns Data” on page 6-44
- “Import Data Using the MATLAB® Interface to SQLite” on page 6-75

- “Roll Back and Commit Data in a Database” on page 6-13
- “Change the Database Connection Catalog” on page 6-14
- “Create a Table and Add a Column” on page 6-15
- “Run a Custom Database Function” on page 6-48

Input Arguments

`conn` — Database connection

database connection object | SQLite connection object

Database connection, specified as a database connection object or SQLite connection object created using `database` or `sqlite`.

`sqlquery` — SQL statement

character vector

SQL statement, specified as a character vector. 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`.

Data Types: `char`

`qTimeOut` — Timeout value

scalar

Timeout value, specified as a 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: `'cursorType','scrollable'`

'cursorType' — Cursor type

'scrollable'

Cursor type, specified as 'scrollable' that creates a scrollable cursor. For details, see "Importing Data Using a Scrollable Cursor" on page 6-59.

Data Types: char

'maxRows' — Maximum number of rows to return

scalar

Maximum number of rows to return, specified as a nonnegative scalar value. Before exec runs the SQL query, the maximum number of rows to return is set using this name-value pair argument. For details about this option and other memory management options, see "Managing Memory to Import Data" on page 6-50.

Data Types: double

Output Arguments

curs — Database cursor

database cursor object

Database cursor, returned as a database cursor object. The properties of this object are different depending on the database connection object.

For a JDBC/ODBC bridge or a JDBC driver database connection, the cursor object has the following properties.

Property	Description
Data	Resulting data after executing fetch.
DatabaseObject	Database connection object or database.ODBCConnection object that opened the cursor object.
RowLimit	Number of rows to fetch at a time.
SQLQuery	SQL statement to execute.
Message	Error messages generated from executing the SQL statement. If this property is empty, then the SQL statement executed successfully.

Property	Description

	To throw error messages to the Command Window, use `setdbprefs`. Enter this code:
	`setdbprefs('Errorhandling','report');`
	`curs = exec(conn,'select * invalidtablename')`
	To store error messages in `curs.Message` instead of sending them to the Command Window, use `setdbprefs`. Enter this code:
	`setdbprefs('Errorhandling','store');`

`Type`	Database cursor object.
`ResultSet`	Java result set object.
`Cursor`	Internal Java representation of a cursor object.
`Statement`	Java statement object.
`Fetch`	Internal Java representation of the fetched data.
`Scrollable`	Logical value that identifies 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 a native ODBC connection, the cursor object has only these properties from the previous list: `Data`, `RowLimit`, `SQLQuery`, `Message`, `Type`, `Statement`, `Scrollable`, and `Position`.

To import data from the cursor object, run `fetch`. To get properties of the cursor object, use `resultset` and `rsmd`, and access the `Statement` property.

A cursor object stays open until you close it using the `close` function.

Limitations

The name-value pair argument `'maxRows'` has these limitations:

- The native ODBC interface is not supported 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.

More About

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.
  ```
- You can experience issues with text field formats in the Microsoft SQL Server database management system. The workarounds are to:
  - Convert fields of format `NVARCHAR`, `TEXT`, `NTEXT`, and `VARCHAR` to `CHAR` in the database.
  - Use `sqlquery` to convert data to `VARCHAR`. For example, run a `sqlquery` statement of the form `'select convert(varchar(20),field1) from table1'`.
- 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#')
  ```
- "Managing Memory to Import Data" on page 6-50

- “Importing Data Using a Scrollable Cursor” on page 6-59
- “Import Large Data Using Paging” on page 6-69
- “Working with the MATLAB Interface to SQLite” on page 2-6
- “Data Retrieval Restrictions” on page 1-8

See Also

`close` | `database` | `fetch` | `setdbprefs`

Introduced before R2006a

exportedkeys

Retrieve information about exported foreign keys

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`

Column	Description	Value
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`

Introduced before R2006a

fastinsert

Add MATLAB data to database table

To export MATLAB data into a database, use these functions: `fastinsert`, `datainsert`, and `insert`. For maximum performance, use `datainsert`.

For other differences among these functions, see “Inserting Data Using the Command Line” on page 2-197.

Syntax

`fastinsert(conn,tablename,colnames,data)`

Description

`fastinsert(conn,tablename,colnames,data)` exports records from the MATLAB variable `data` into new rows in an existing database table `tablename` and in existing columns `colnames` using the connection `conn`. You do not specify the type of data you are exporting; the data is exported in its current MATLAB format.

Examples

Insert a Table Record Using Native ODBC

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 `dbtoolboxdemo` with `admin` as the user name and password. This database contains the table `productTable` with these columns:

- `productNumber`
- `stockNumber`
- `supplierNumber`
- `unitCost`
- `productDescription`

```
conn = database.ODBCConnection('dbtoolboxdemo','admin','admin')

conn = 

  connection with properties:

      Instance: 'dbtoolboxdemo'
      UserName: 'admin'
       Message: []
        Handle: [1x1 database.internal.ODBCConnectHandle]
       TimeOut: 0
    AutoCommit: 0
          Type: 'ODBCConnection Object'
```

`conn` has an empty `Message` property, which means a successful connection.

Select and display the data from the `productTable`. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
sqlquery = 'select * from productTable';

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

ans = 

    productNumber    stockNumber    supplierNumber    unitCost    productDescription
    -------------    -----------    --------------    --------    ------------------
    ...
     6               400876         1004               8          'Sail Boat'
     3               400999         1009              17          'Slinky'
    10               888652         1006              24          'Teddy Bear'
```

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

```
close(curs)
```

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 the cell array `data` that contains these values:

- `productNumber` equal to 11

- `stockNumber` equal to 500565
- `supplierNumber` equal to 1010
- `unitCost` equal to $20
- `productDescription` equal to `'Cooking Set'`

Then, convert the cell array to the table `data_table`.

```
data = {11,500565,1010,20,'Cooking Set'};
data_table = cell2table(data,'VariableNames',colnames)
```

```
data_table = 

    productNumber    stockNumber    supplierNumber    unitCost    productDescription
    -------------    -----------    --------------    --------    ------------------
    11               500565         1010              20          'Cooking Set'
```

Insert the table data into the `productTable`.

```
fastinsert(conn,tablename,colnames,data_table)
```

Display the data from the `productTable` again.

```
sqlquery = 'select * from productTable';

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

```
ans = 

    productNumber    stockNumber    supplierNumber    unitCost    productDescription
    -------------    -----------    --------------    --------    ------------------
    ...
     3               400999         1009              17          'Slinky'
    10               888652         1006              24          'Teddy Bear'
    11               500565         1010              20          'Cooking Set'
```

A new row appears in the `productTable` with the data from `data_table`.

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

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

Insert a Record

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 `dbtoolboxdemo` with `admin` as the user name and password. This database contains the table `inventoryTable` with these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

```
conn = database('dbtoolboxdemo','admin','admin');
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Display the data in the `inventoryTable` table before insertion. 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);
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...'
```

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

```
close(curs)
```

Assign the data for insertion to the cell array `data`. The data is:

- `productNumber` is 7777
- `Quantity` is 100
- `Price` is 50.00
- `inventoryDate` is the date of the current moment

```
data = {7777,100,50.00,datestr(now,'yyyy-mm-dd HH:MM:SS')};
```

Create a cell array containing these column names: `productNumber`, `Quantity`, `Price`, `inventoryDate`.

```
tablename = 'inventoryTable';
colnames = {'productNumber','Quantity','Price','inventoryDate'};
```

Insert the data into the `inventoryTable`.

```
fastinsert(conn,tablename,colnames,data)
```

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

```
sqlquery = 'select * from inventoryTable';

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

```
ans =

    ...
    [   12]    [ 1278]    [       0]    '2010-10-29 18:17...'
    [   13]    [ 1700]    [14.5000]    '2009-05-24 10:58...'
    [7777]    [  100]    [      50]    '2014-10-23 10:01...'
```

The last row contains the inserted data.

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

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

Insert Multiple Records

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 `dbtoolboxdemo` with `admin` as the user name and password. This database contains the table `inventoryTable` with these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

```
conn = database('dbtoolboxdemo','admin','admin');
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Display the data in the `inventoryTable` table before insertion. 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);
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...'
```

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

```
close(curs)
```

Assign multiple rows of data to the cell array `data`. Each row contains data for `productNumber`, `Quantity`, and `Price`. The first row data for insertion is `productNumber` equals 7778, `Quantity` equals 125, and `Price` equals 23.00.

```
data = {7778,125,23.00,datestr(now,'yyyy-mm-dd HH:MM:SS');...
        7779,1160,14.7,datestr(now,'yyyy-mm-dd HH:MM:SS');...
        7780,150,54.5,datestr(now,'yyyy-mm-dd HH:MM:SS')};
```

Create a cell array containing these column names:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

```
tablename = 'inventoryTable';
```

```
colnames = {'productNumber','Quantity','Price','inventoryDate'};
```

Insert the data into the `inventoryTable`.

```
fastinsert(conn,tablename,colnames,data)
```

For details about inserting data in bulk, see these sample files for different database vendors that demonstrate bulk insert:

- `*matlabroot*/toolbox/database/dbdemos/mssqlserverbulkinsert.m`
- `*matlabroot*/toolbox/database/dbdemos/mysqlbulkinsert.m`
- `*matlabroot*/toolbox/database/dbdemos/oraclebulkinsert.m`

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

```
sqlquery = 'select * from inventoryTable';

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

```
ans =

    ...
    [  13]    [ 1700]    [14.5000]    '2009-05-24 10:58...'
    [7778]    [  125]    [     23]    '2014-10-23 10:21...'
    [7779]    [ 1160]    [14.7000]    '2014-10-23 10:21...'
    [7780]    [  150]    [54.5000]    '2014-10-23 10:21...'
```

The last three rows contain the inserted data.

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

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

Import Records, Perform Calculations, and Export Data

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 `dbtoolboxdemo` with `admin` as the user name and password. This database contains the tables `salesVolume` and `yearlySales`.

```
conn = database('dbtoolboxdemo','admin','admin');
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Use `setdbprefs` to set the format for retrieved data to `numeric`.

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

Import 10 rows of data from the `March` column in 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);
```

Assign the data to the MATLAB workspace variable `AA`.

```
AA = curs.Data
```

```
AA =

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

Calculate the sum of the `March` sales and assign the result to the variable `sumA`.

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

```
sumA =

   14606
```

Assign the month and sum of sales to a cell array to export to a database. Put the month in the first cell of `data`.

```
data(1,1) = {'March'}
```

```
data =
  'March'
```

Put the sum in the second cell of `data`.

```
data(1,2) = {sumA}
```

```
data =
    'March'    [14606]
```

Define the names of the columns to which to export data. In this example, the column names are `Month` and `salesTotal`, from the `yearlySales` table in the `dbtoolboxdemo` database. Assign the cell array containing the column names to the variable `colnames`.

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

Access the status of the `AutoCommit` database flag. This status determines whether the exported data is automatically committed to the database. If the flag is `off`, you can undo an insert; if it is `on`, data is automatically committed to the database.

```
conn.AutoCommit
```

```
ans =
 on
```

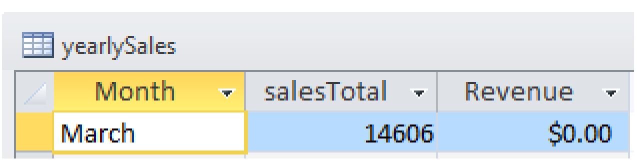
The `AutoCommit` flag is set to `on`, so the exported data is automatically committed to the database.

Use `fastinsert` to export the data into the `yearlySales` table.

```
fastinsert(conn,tablename,colnames,data)
```

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

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


yearlySales

Month	salesTotal	Revenue
March	14606	$0.00

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

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

Insert Numeric Data

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 `dbtoolboxdemo` with `admin` as the user name and password. This database contains the table `salesVolume`. This table contains the column `stockNumber` and columns for the months of the year.

```
conn = database('dbtoolboxdemo','admin','admin');
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

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];
```

Display the data in the `salesVolume` table before insertion. 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);
curs.Data
```

```
ans =

  Columns 1 through 8

    ...
    [400876]    [3000]    [2400]    [1500]    [1500]    [1300]    [1100]    [ 900]
    [400999]    [3000]    [1500]    [1000]    [ 900]    [ 750]    [ 700]    [ 400]
    [888652]    [ NaN]    [ 900]    [ 821]    [ 701]    [ 689]    [ 621]    [ 545]

  Columns 9 through 13

    ...
    [867]    [ 923]    [1100]    [ 4000]    [32000]
```

```
    [350]     [ 500]     [1100]     [ 3000]     [12000]
    [421]     [ 495]     [ 550]     [ 4200]     [12000]
```

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

```
close(curs)
```

Insert `data` using the columns denoted by `colnames` into the `salesVolume` table.

```
tablename = 'salesVolume';
colnames = {'stockNumber','January','February'...
            'March','April','May',...
            'June','July','August',...
            'September','October','November',...
            'December'};

fastinsert(conn,tablename,colnames,data)
```

Display the data in the `salesVolume` table after insertion.

```
curs = exec(conn,'select * from salesVolume');
curs = fetch(curs);
curs.Data

ans =

  Columns 1 through 8

    ...
    [400999]    [3000]    [1500]    [1000]    [ 900]    [ 750]    [ 700]    [ 400]
    [888652]    [ NaN]    [ 900]    [ 821]    [ 701]    [ 689]    [ 621]    [ 545]
    [777666]    [   0]    [ 350]    [ 400]    [ 450]    [ 250]    [ 450]    [ 500]

  Columns 9 through 13

    ...
    [350]     [ 500]     [1100]     [ 3000]     [12000]
    [421]     [ 495]     [ 550]     [ 4200]     [12000]
    [515]     [ 235]     [ 100]     [  300]     [  600]
```

The last row contains the inserted data.

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

```
close(curs)
```

```
close(conn)
```

Insert and Commit Data

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 `dbtoolboxdemo` with `admin` as the user name and password. This database contains the table `inventoryTable` with these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

```
conn = database('dbtoolboxdemo','admin','admin');
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Set the `AutoCommit` flag to `off`.

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

Insert the cell array `data` into the `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 inserted data.

```
commit(conn)
```

Alternatively, commit the data using an SQL `commit` statement with the `exec` function. The cursor object `curs` contains the executed query.

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

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

```
close(curs)
```

```
close(conn)
```

Insert Boolean Data

Connect to the data source `dbtoolboxdemo`. This data source identifies a Microsoft Access database. This database contains the table `invoice` with these columns:

- `InvoiceNumber`
- `InvoiceDate`
- `productNumber`
- `Paid`
- `Receipt`

```
conn = database('dbtoolboxdemo','admin','admin');
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Display the data in the `invoice` table before insertion.

```
curs = exec(conn,'select * from invoice');
curs = fetch(curs);
curs.Data
```

```
ans =

    ...
    [41011]    '2011-12-12 00:00...'    [ 8]    [1]    [1920474x1 int8]
    [61178]    '2012-01-15 00:00...'    [ 9]    [0]    [2378330x1 int8]
    [62145]    '2012-01-23 00:00...'    [10]    [1]    [ 492314x1 int8]
```

Create `data` as a structure containing the invoice number `2101` and the Boolean data `1` to signify paid. Boolean data is represented as the MATLAB type `logical`. Here, assume that the receipt image is missing.

```
data.InvoiceNumber{1} = 2101;
data.InvoiceDate{1} = datestr(now,'yyyy-mm-dd HH:MM:SS');
data.productNumber{1} = 11;
data.Paid{1} = logical(1);
```

Insert the paid invoice data into the `invoice` table with column names `colnames`.

```
colnames = {'InvoiceNumber';'InvoiceDate';'productNumber';'Paid'};
```

```
tablename = 'invoice';
```

```
fastinsert(conn,tablename,colnames,data)
```

View the new record in the database to verify that the `Paid` field is Boolean. In some databases, the MATLAB logical value `0` shows as a Boolean `false`, `No`, or a cleared check box.

```
curs = exec(conn,'select * from invoice');
curs = fetch(curs);
curs.Data
```

```
ans = 

    ...
    [61178]    '2012-01-15 00:00...'    [ 9]    [0]    [2378330x1 int8]
    [62145]    '2012-01-23 00:00...'    [10]    [1]    [ 492314x1 int8]
    [ 2101]    '2014-10-23 11:14...'    [11]    [1]                  []
```

The last row contains the Boolean data `1`.

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 6-22
- "Export Multiple Records from the MATLAB Workspace" on page 6-27
- "Export Data Using Bulk Insert" on page 6-31
- "Replace Existing Data in a Database" on page 6-25
- "Roll Back Data After Updating a Record" on page 6-19

Input Arguments

`conn` — Database connection

database connection object

Database connection, specified as a database connection object created using `database`.

`tablename` — Database table name

character vector

Database table name, specified as a character vector denoting the name of a table in your database.

Data Types: `char`

`colnames` — Database table column names

cell array of character vectors

Database table column names, specified as a cell array of one or more character vectors to denote the columns in the existing database table `tablename`.

Example: `{'col1','col2','col3'}`

Data Types: `cell`

`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, field names in the structure must match `colnames`. If `data` is a table or a dataset array, 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 insert dates and timestamps with the native ODBC interface, use the format `'YYYY-MM-DD HH:MM:SS.MS'`.

To reduce conversion time, convert dates to serial date numbers using `datenum` before calling `fastinsert`.

More About

Tips

- The status of the `AutoCommit` flag determines whether `fastinsert` automatically commits the data to the database. Use `get` to view the `AutoCommit` flag status for the connection and use `set` to change it. Use `commit` or issue an SQL COMMIT statement using `exec` to commit the data to the database. Use `rollback` or issue an SQL ROLLBACK statement using `exec` to roll back the data.

- 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.

- “Inserting Data Using the Command Line” on page 2-197
- “Connecting to a Database Using the Native ODBC Interface” on page 3-18
- “Getting Started with Visual Query Builder” on page 5-2

See Also

`close` | `commit` | `database` | `exec` | `get` | `insert` | `logical` | `rollback` | `set` | `update`

Introduced before R2006a

fetch

Import data into MATLAB workspace from cursor object or from execution of SQL statement

Investigate the data and its structure by passing a database cursor object to fetch. Or, import and view data without running `exec` using the database connection object instead.

Import data from a SQLite database file immediately using a SQLite connection object of the MATLAB interface to SQLite.

To import data interactively, use the Database Explorer app.

Caution: Leaving cursor and connection objects open or overwriting open objects can result in unexpected behavior. Once you finish working with these objects, you must close them using `close`.

Syntax

```
curs = fetch(curs)
curs = fetch(curs,rowlimit)
curs = fetch(curs,Name,Value)
curs = fetch(curs,rowlimit,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 into the cursor object `curs` from the open SQL cursor object `curs`.

`curs = fetch(curs,rowlimit)` imports rows of data up to the maximum number of rows `rowlimit`.

`curs = fetch(curs,Name,Value)` imports rows of data using a scrollable cursor.

`curs = fetch(curs,rowlimit,Name,Value)` imports rows of data up to the maximum number of rows `rowlimit` using a scrollable cursor.

`results = fetch(conn,sqlquery)` executes the SQL statement `sqlquery` and imports all rows of data in batches for the open database connection or SQLite connection `conn`. `results` contains the resulting data.

`results = fetch(conn,sqlquery,fetchbatchsize)` imports all rows of data in batches of a specified number of rows `fetchbatchsize` at a time.

`results = fetch(conn,sqlquery,rowlimit)` imports rows of data up to the maximum number of rows `rowlimit` using the SQLite connection `conn`.

Examples

Import All Data Using the Native ODBC Interface and Cursor Object

Create a connection `conn` using the native ODBC interface and the `dbtoolboxdemo` data source.

```
conn = database.ODBCConnection('dbtoolboxdemo','admin','admin')
```

```
conn = 

  connection with properties:

      Instance: 'dbtoolboxdemo'
      UserName: 'admin'
       Message: []
        Handle: [1x1 database.internal.ODBCConnectHandle]
       TimeOut: 0
    AutoCommit: 0
          Type: 'ODBCConnection Object'
```

`conn` has an empty `Message` property, which means a successful connection.

Working with the `dbtoolboxdemo` data source, use `fetch` to import all data into the `database.ODBCCursor` object `curs`. Store the data in a cell array contained in the cursor object property `curs.Data`.

```
curs = exec(conn,'select productDescription from productTable');
curs = fetch(curs)

curs =

  cursor with properties:

         Data: {10x1 cell}
     RowLimit: 0
     SQLQuery: 'select productDescription from productTable'
      Message: []
         Type: 'ODBCCursor Object'
    Statement: [1x1 database.internal.ODBCStatementHandle]
```

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

View the contents of the `Data` element 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'
```

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

```
close(curs)
```

Import Specified Rows Using the Cursor Object

Working with the `dbtoolboxdemo` data source, use the `rowlimit` argument to retrieve only the first three rows of data. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
sqlquery = 'select productdescription from producttable';
```

```
curs = exec(conn,sqlquery);
curs = fetch(curs,3)

curs =

  cursor with properties:

        Attributes: []
              Data: {3x1 cell}
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: 'select productdescription from producttable'
           Message: []
              Type: 'Database Cursor Object'
         ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
            Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
         Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
             Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

View the data.

```
curs.Data

ans =

    'Victorian Doll'
    'Train Set'
    'Engine Kit'
```

Rerun the `fetch` function to return the second three rows of data.

```
curs = fetch(curs,3);
```

View the data.

```
curs.Data

ans =

    'Painting Set'
    'Space Cruiser'
    'Building Blocks'
```

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

```
close(curs)
```

Import Data Iteratively Using the Cursor Object

Working with the `dbtoolboxdemo` data source, use the `rowlimit` argument to retrieve the first two rows of data. To retrieve two rows at a time, rerun the import using a `while` loop. Continue until you have retrieved all the data, which occurs when `curs.Data` returns `'No Data'`. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
sqlquery = 'select productdescription from producttable';

curs = exec(conn,sqlquery);

% Initialize rowlimit
rowlimit = 2

% Check for more data. Retrieve and display all data.
while ~strcmp(curs.Data,'No Data')
 curs = fetch(curs,rowlimit);
 curs.Data(:)
end
```

```
rowlimit =

     2


ans =

    'Victorian Doll'
    'Train Set'


ans =

    'Engine Kit'
    'Painting Set'


ans =

    'Space Cruiser'
    'Building Blocks'
```

```
ans =

    'Tin Soldier'
    'Sail Boat'


ans =

    'Slinky'
    'Teddy Bear'

ans =
    'No Data'
```

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

```
close(curs)
```

Import Data with an Absolute Position Offset Using the Scrollable Cursor

Connect to the MySQL database using the native ODBC interface. This code assumes that you are connecting to a data source named `MySQL` with user name `username` and password `pwd`. This database contains a table called `productTable`. This table contains 15 records, where each record represents one product.

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

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'`.

```
curs = exec(conn,'select * from productTable order by productNumber',...
            'cursorType','scrollable');
```

Import the last five products in the data set using the absolute position offset `11`.

```
curs = fetch(curs,'absolutePosition',11);
```

Display the data for the five products.

```
curs.Data
```

```
ans =
```

```
[11]     [408143]     [1004]     [      11]     'Convertible'
[12]     [210456]     [1010]     [      22]     'Hugsy'
[13]     [470816]     [1012]     [16.5000]     'Pancakes'
[14]     [510099]     [1011]     [      19]     'Shawl'
[15]     [899752]     [1011]     [      20]     'Snacks'
```

The columns in `curs.Data` are:

- Product number
- Stock number
- Supplier number
- Unit cost
- Product description

After calling `fetch`, the position of the cursor is located after the data set.

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

```
close(curs)
```

Import Data with a Row Limit Using the Scrollable Cursor

Connect to the MySQL database using the native ODBC interface. This code assumes that you are connecting to a data source named `MySQL` with user name `username` and password `pwd`. This database contains a table called `productTable`. This table contains 15 records, where each record represents one product.

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

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'`.

```
curs = exec(conn,'select * from productTable order by productNumber',...
            'cursorType','scrollable');
```

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 `3` to import data starting from the third product in the data set.

```
curs = fetch(curs,2,'absolutePosition',3);
```

Display the data for the two products.

```
curs.Data
```

```
ans =

    [3]    [400999]    [1009]    [17]    'Slinky'
    [4]    [400339]    [1008]    [21]    'Space Cruiser'
```

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 =

     3
```

The position of the cursor stays at the absolute position offset `3`.

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

```
close(curs)
```

Import Data with Different Formats Using the Cursor Object

Import data that includes a `BOOLEAN` field. Specify the format `cellarray` for the retrieved data using the `setdbprefs` function. The cursor object `curs` contains the executed query. Import the data from the executed query using the `fetch` function.

```
curs = exec(conn,['select InvoiceNumber, '...
'Paid from Invoice']);
setdbprefs('DataReturnFormat','cellarray')
curs = fetch(curs,5);
A = curs.Data
```

```
A =
```

```
    [ 2101]    [0]
    [ 3546]    [1]
    [33116]    [1]
    [34155]    [0]
    [34267]    [1]
```

View the class of the second column of A.

```
class(A{1,2})
```

```
ans =
logical
```

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

```
close(curs)
```

Import Data Using the Database Connection Object

Working with the `dbtoolboxdemo` data source, import the `productDescription` column from `productTable`. Set the data return format to `'cellarray'` using `setdbprefs`.

```
setdbprefs('DataReturnFormat','cellarray')
sqlquery = 'select productdescription from productTable';

results = fetch(conn,sqlquery)
```

```
results =

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

View the size of the cell array into which the results were returned.

```
size(results)
```

```
ans =

    10     1
```

Close the database connection.

```
close(conn)
```

Import Data with `fetchbatchsize` Using the Database Connection Object

Working with the `dbtoolboxdemo` data source, import the `productDescription` column from the `productTable` using the `fetchbatchsize` argument.

```
setdbprefs('DataReturnFormat','cellarray')
sqlquery = 'select productdescription from productTable';
fetchbatchsize = 5;

results = fetch(conn,sqlquery,fetchbatchsize);
```

`fetch` returns all the data by importing it in batches of five rows at a time.

Close the database connection.

```
close(conn)
```

Import Data Using the MATLAB® Interface to SQLite

Create a SQLite connection `conn` to an existing SQLite database file `tutorial.db`.

```
dbfile = 'tutorial.db';

conn = sqlite(dbfile);
```

Import the `productDescription` column from the `productTable` by using the `rowlimit` argument.

```
sqlquery = 'select productdescription from productTable';
rowlimit = 5;

results = fetch(conn,sqlquery,rowlimit);
```

`results` returns five rows of data.

Close the SQLite connection.

```
close(conn)
```

- “Import Data from Databases into MATLAB” on page 6-4
- “Import Data Incrementally Using the Cursor Object” on page 6-53
- “Import Data Using a Scrollable Cursor with a Relative Position Offset” on page 6-66
- “Retrieve Image Data Types” on page 6-37
- “Display Information About Imported Data” on page 6-56
- “Import Data Using the MATLAB® Interface to SQLite” on page 6-75

Input Arguments

`curs` — Database cursor
database cursor object

Database cursor, specified as an open SQL database cursor object created using `exec`.

`conn` — Database connection
database connection object | SQLite connection object

Database connection, specified as a database connection object or SQLite connection object created using `database` or `sqlite`.

`sqlquery` — SQL statement
character vector

SQL statement, specified as a character vector.

Data Types: `char`

`rowlimit` — Row limit
scalar

Row limit, specified as a scalar denoting the number of rows of data to import from the open SQL cursor object `curs`.

If `rowlimit` is `0`, `fetch` returns all rows of data.

Data Types: `double`

`fetchbatchsize` — Fetch batch size
scalar

Fetch batch size, specified as a scalar denoting 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. If `fetchbatchsize` is not provided, a default value of `'FetchBatchSize'` is used. `'FetchBatchSize'` is set using `setdbprefs`.

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: `'absolutePosition',5`

`'absolutePosition'` — Absolute position offset
scalar

Absolute position offset, specified as a scalar to denote 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 "Importing Data Using a Scrollable Cursor" on page 6-59.

Data Types: `double`

`'relativePosition'` — Relative position offset
scalar

Relative position offset, specified as a scalar to denote 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 "Importing Data Using a Scrollable Cursor" on page 6-59.

Data Types: `double`

Output Arguments

`curs` — Database cursor

database cursor object

Database cursor, returned as a database cursor object populated with fetched data in the `Data` property. You can specify the output data format in the `Data` property using `setdbprefs`.

`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 `setdbprefs`. The result data contains all rows of data from the executed SQL statement.

If `conn` is a SQLite connection, `results` is a cell array only. The cell array contains only one of these data types: `DOUBLE`, `INT64`, and `CHAR`. If `NULL`s 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 a Driver and Data Source" on page 2-16.

More About

Tips

- The order of records in your database does not remain constant. Sort data using the SQL `ORDER BY` command in your `sqlquery` statement.
- When working with a JDBC or JDBC/ODBC bridge connection, running `fetch` on the cursor object returns a new object of type cursor. The cursor object points to the same underlying Java objects as the input cursor. For best results, overwrite the input cursor object. Overwriting this object means that only one open cursor object exists, which consumes less memory than multiple open cursor objects.

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

Then, close this one object. Creating a different variable for the output cursor object creates two objects pointing to the same underlying Java statement and result set objects.

With a native ODBC connection established using `database.ODBCConnection`, 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 the following usages:

- `curs = fetch(curs)`
- `fetch(curs)`
- `curs2 = fetch(curs)`

- "Importing Data Using a Scrollable Cursor" on page 6-59
- "Managing Memory to Import Data" on page 6-50
- "Connecting to a Database Using the Native ODBC Interface" on page 3-18
- "Working with the MATLAB Interface to SQLite" on page 2-6
- "Preference Settings for Large Data Import" on page 5-19

See Also

`close` | `database` | `exec` | `fetchmulti` | `setdbprefs`

Introduced before R2006a

fetchmulti

Import data from multiple resultsets

To retrieve multiple resultsets, use `exec` with a `sqlquery` statement. `sqlquery` can contain two or more SELECT statements or run a stored procedure consisting of two or more SELECT statements. Then, use `fetchmulti` to retrieve the data in each resultset.

Syntax

```
curs = fetchmulti(curs)
```

Description

`curs = fetchmulti(curs)` imports data from an open SQL cursor object `curs` that contains multiple resultsets into the object `curs`.

Examples

Retrieve Multiple Resultsets

Create a database connection `conn` to the Microsoft SQL Server database. To connect without Windows authentication using the native ODBC interface, connect to the database with the ODBC data source name. Here, this code assumes that you are connecting to a data source named `MS SQL Server` with user name `username` and password `pwd`.

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

Select all data from two tables using two SELECT statements in `sqlquery`.

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

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

Import data from the two resultsets.

```
curs = fetchmulti(curs)
```

```
curs =

  cursor with properties:

         Data: {{23x4 cell}  {15x5 cell}}
     RowLimit: 0
     SQLQuery: 'SELECT * FROM inventoryTable; SELECT * FROM productT…'
      Message: []
         Type: 'ODBCCursor Object'
    Statement: [1x1 database.internal.ODBCStatementHandle]
```

`curs.Data` is a cell array consisting of cell arrays, tables, structures, or numeric matrices as specified in `setdbprefs`. The data type is the same for all resultsets.

`curs.Data` contains the data from both resultsets. The first cell array contains data from the first SELECT statement. The second cell array contains data from the second SELECT statement.

Display the data from both tables.

```
resultset_one = curs.Data{1,1}
resultset_two = curs.Data{1,2}
```

```
resultset_one =

    [ 1.00]    [ 1700.00]    [ 14.50]    '2014-10-20 00:00:…'
    [ 2.00]    [ 1200.00]    [  9.30]    '2014-10-20 00:00:…'
    [ 3.00]    [  356.00]    [ 17.20]    '2014-10-20 00:00:…'
    ...

resultset_two =

  Columns 1 through 4

    [ 9.00]    [125970.00]    [1003.00]    [13.00]
    [ 8.00]    [212569.00]    [1001.00]    [ 5.00]
    [ 7.00]    [389123.00]    [1007.00]    [16.00]
    ...

  Column 5

    'Victorian Doll'
    'Train Set'
```

```
    'Engine Kit'
    ...
```

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

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

- “Call a Stored Procedure That Returns Data” on page 6-44
- “Import Data from Databases into MATLAB” on page 6-4
- “Display Information About Imported Data” on page 6-56

Input Arguments

`curs` — Database cursor
database cursor object

Database cursor, specified as an open SQL database cursor object created using `exec`.

Output Arguments

`curs` — Database cursor
database cursor object

Database cursor, returned as a database cursor object populated with fetched data in the `Data` property. You can specify the output data format in the `Data` property using `setdbprefs`.

More About

- “Create Queries with Characters and Variables” on page 6-8

See Also

`database` | `exec` | `fetch` | `setdbprefs`

Introduced in R2006b

get

Retrieve object properties

Compatibility

`driver` and `drivermanager` have been removed.

`resultset` and `rsmd` will be removed in a future release.

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 database connection object.

Establish connection `conn` to a MySQL database with user name `username` and password `pwd`.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Construct a database metadata object `dbmeta` using the database connection object `conn`.

```
dbmeta = dmd(conn);
```

Retrieve the properties of `dbmeta` and assign them to MATLAB variable `v`.

```
v = get(dbmeta)
```

```
v = 

                 AllProceduresAreCallable: 1
                   AllTablesAreSelectable: 1
    DataDefinitionCausesTransactionCommit: 1
      DataDefinitionIgnoredInTransactions: 0
               DoesMaxRowSizeIncludeBlobs: 0
                                 Catalogs: {8x1 cell}
                         CatalogSeparator: '.'
                              CatalogTerm: 'DATABASE'
                      DatabaseProductName: 'ACCESS'
                   DatabaseProductVersion: '04.00.0000'
              DefaultTransactionIsolation: 2
                       DriverMajorVersion: 2
                       DriverMinorVersion: 1
                               DriverName: 'JDBC-ODBC Bridge (ACEODBC.DLL)'
                            DriverVersion: '2.0001 (Microsoft Access database engine)'
                      ExtraNameCharacters: '~@#$%^&*_-+=\}{"';:?/><,.![]|'
                    IdentifierQuoteString: '`'
                         IsCatalogAtStart: 1
                   MaxBinaryLiteralLength: 255
                     MaxCatalogNameLength: 260
                     MaxCharLiteralLength: 255
                      MaxColumnNameLength: 64
                      MaxColumnsInGroupBy: 10
                        MaxColumnsInIndex: 10
                      MaxColumnsInOrderBy: 10
                       MaxColumnsInSelect: 255
                        MaxColumnsInTable: 255
                           MaxConnections: 64
                      MaxCursorNameLength: 64
                           MaxIndexLength: 255
                   MaxProcedureNameLength: 64
                               MaxRowSize: 4052
                      MaxSchemaNameLength: 0
                       MaxStatementLength: 65000
                            MaxStatements: 0
                       MaxTableNameLength: 64
                        MaxTablesInSelect: 16
                        MaxUserNameLength: 0
                         NumericFunctions: [1x73 char]
                            ProcedureTerm: 'QUERY'
                                  Schemas: {}
                               SchemaTerm: ''
                       SearchStringEscape: '\'
                              SQLKeywords: [1x255 char]
                          StringFunctions: [1x91 char]
               StoresLowerCaseIdentifiers: 0
         StoresLowerCaseQuotedIdentifiers: 0
               StoresMixedCaseIdentifiers: 0
         StoresMixedCaseQuotedIdentifiers: 1
```

```
        StoresUpperCaseIdentifiers: 0
  StoresUpperCaseQuotedIdentifiers: 0
                   SystemFunctions: ''
                        TableTypes: {18x1 cell}
                 TimeDateFunctions: [1x111 char]
                          TypeInfo: {16x1 cell}
                               URL: 'jdbc:odbc:tutorial2'
                          UserName: 'admin'
              NullPlusNonNullIsNull: 0
                NullsAreSortedAtEnd: 0
              NullsAreSortedAtStart: 0
                 NullsAreSortedHigh: 0
                  NullsAreSortedLow: 1
              UsesLocalFilePerTable: 0
                     UsesLocalFiles: 1
```

Display the contents of the `Catalogs` property of `v`.

```
v.Catalogs
```

```
ans =
    'D:\matlab\toolbox\database\dbdemos\db1'
    'D:\matlab\toolbox\database\dbdemos\origtutorial'
    'D:\matlab\toolbox\database\dbdemos\tutorial'
    'D:\matlab\toolbox\database\dbdemos\tutorial1'
```

Close the connection.

```
close(conn)
```

Get the AutoCommit Flag Status

Retrieve the `'AutoCommit'` property of the database connection object.

Establish connection `conn` to a MySQL database with user name `username` and password `pwd`.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Check the status of the `'AutoCommit'` property for the database connection `conn`.

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

```
v =
 on
```

Close the connection.

```
close(conn)
```

- "Display Database Metadata" on page 6-39
- "Display Information About Imported Data" on page 6-56

Input Arguments

`object` — Database Toolbox object

database connection object | cursor object | ...

Database Toolbox object, specified as the following allowable objects:

- Database connection object, which is created using `database`
- Cursor object, which is created using `exec` or `fetch`
- Database metadata object, which is created using `dmd`
- Resultset object, which is created using `resultset`
- Resultset metadata object, which is created using `rsmd`

`property` — Property of Database Toolbox object

character vector

Property of the Database Toolbox object, specified as a character vector.

For database connection objects, see this table for the available property names and returned values.

Database 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.
`'Catalog'`	Name of the catalog 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`.
`'Handle'`	JDBC connection object.

Database Connection Object Property	Description
'Instance'	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'	1 if the database is read only; 0 if the database is writable.
'TimeOut'	Number of seconds that the driver waits while trying to establish a database connection before throwing an error.
'Type'	Object type, specifically 'Database Object'.
'URL'	For JDBC connections only, the JDBC URL object jdbc:*subprotocol*:*subname*, as specified by database.
'UserName'	User name required to connect to a given database, as specified by database.
'Warnings'	Warnings returned 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
'Attributes'	Cursor attributes. This field is always empty. Use the attr function to retrieve cursor attributes.
'Data'	Data in the cursor object data element (the query results).
'DatabaseObject'	Information about a given database object.
'RowLimit'	Maximum number of rows returned by fetch, as specified by set.
'SQLQuery'	SQL statement for a cursor, as specified by exec.
'Message'	Error message returned from exec or fetch.
'Type'	Object type, specifically 'Database Cursor Object'.
'ResultSet'	Handle to Java resultset object.
'Cursor'	Handle to Java cursor object.
'Statement'	Handle to Java statement object.

Cursor Object Property	Description
'Fetch'	0 for a cursor created using exec; fetchTheData for a cursor created using fetch.
'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'}
'DatabaseProductNa	Database vendor name	'ACCESS'
'DatabaseProductVe	Database version number	'03.50.0000'
'DriverName'	Name of the JDBC or ODBC driver	'JDBC-ODBC Bridge (odbcjt32.dll)'
'MaxColumnNameLeng	Maximum length of the database column name	64
'MaxColumnsInOrder	Maximum number of database columns for sorting the data	10
'URL'	JDBC database URL for establishing a connection	'jdbc:odbc:dbtoolboxdemo'

For resultset objects, see this table for the available property names and returned values.

Resultset Object Property	Description	Example of Value
'CursorName'	Internal Java cursor object name	{'SQL_CUR92535700x' 'SQL_CUR92535700x'}

Resultset Object Property	Description	Example of Value
'MetaData'	Information about the cursor object	{1x2 cell}
'Warnings'	Query execution warnings	{[] []}

For resultset metadata objects, see this table for the available property names and returned values.

Resultset Metadata Object Property	Description	Example of Value
'CatalogName'	Database catalog name	{'toystore' 'dbo'}
'ColumnCount'	Number of columns in the resultset	2
'ColumnName'	Column names in the resultset	{'Calc_Date' 'Avg_Cost'}
'ColumnTypeNar	Database column data types	{'TEXT' 'LONG'}
'isNullable'	Whether database column can contain NULL values	{[1] [1]}
'isReadOnly'	Whether database column is read only	{[0] [0]}
'TableName'	The table name where the column resides	{'' ''}

When `CatalogName` and `TableName` contain the value `{'' ''}`, databases do not return metadata for catalog and table names.

Data Types: `char`

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.

More About

Tips

- For database connection objects and cursor objects, you can use the native ODBC interface with this function. For details about establishing a connection using the native ODBC interface, see `database`.

See Also

`close` | `columns` | `database` | `dmd` | `exec` | `fetch` | `getdatasources` | `resultset` | `rows` | `rsmd` | `set`

Introduced before R2006a

getdatasources

Return names of ODBC and JDBC data sources on system

Syntax

```
d = getdatasources
```

Description

`d = getdatasources` returns the names of valid ODBC and JDBC data sources on the system as a cell array `d` of character vectors. The function gets the names of ODBC data sources from the `ODBC.INI` file located in the folder returned by running:

```
myODBCdir = getenv('WINDIR')
```

`d` is empty when the `ODBC.INI` file is valid, but no data sources are defined. `d` equals `-1` when the `ODBC.INI` file cannot be opened.

The function also retrieves the names of data sources that are in the system registry but not in the `ODBC.INI` file.

If you do not have write access to `myODBCdir`, the results of `getdatasources` may not include data sources that you recently added. In this case, specify a temporary, writable, output folder via the preference `TempDirForRegistryOutput`. For details about this preference, see `setdbprefs`.

`getdatasources` gets the names of JDBC data sources from the file that you define using `setdbprefs` or the Define JDBC data sources dialog box.

Examples

Get the names of databases on your system.

```
d = getdatasources
d =
    'MS Access Database'  'dbtoolboxdemo'
```

More About

- “Connecting to a Database” on page 2-191
- “Configuring a Driver and Data Source” on page 2-16

See Also

`database` | `get` | `setdbprefs`

Introduced before R2006a

importedkeys

Return information about imported foreign keys

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`

Introduced before R2006a

indexinfo

Return indices and statistics for database tables

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`

Column	Description	Value
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`

Introduced before R2006a

insert

Add MATLAB data to database tables

To export MATLAB data into a database, use these functions: `insert`, `datainsert`, and `fastinsert`. For maximum performance, use `datainsert`. If you connect to the database using a JDBC driver or the JDBC/ODBC bridge, `insert` has the same functionality as `fastinsert`.

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

For other differences among these functions, see “Inserting Data Using the Command Line” on page 2-197.

Syntax

```
insert(conn,tablename,colnames,data)
```

Description

`insert(conn,tablename,colnames,data)` exports records from the MATLAB variable `data` into new rows in an existing database table `tablename` using the connection `conn`.

Examples

Insert a Table Record Using Native ODBC

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 `dbtoolboxdemo` with `admin` as the user name and password. This database contains the table `productTable` with these columns:

- `productNumber`
- `stockNumber`

- `supplierNumber`
- `unitCost`
- `productDescription`

```
conn = database.ODBCConnection('dbtoolboxdemo','admin','admin');
```

Select and display the data from the `productTable` table. The cursor object `curs` 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 the data for the insert 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 the Contents of a Cell Array

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 `dbtoolboxdemo` with blank user name and password. This database contains the table `yearlySales` that contains these columns: `Month`, `salesTotal`, and `Revenue`.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Select and display the data from the `yearlySales` table. The cursor object `curs` 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 the insert in a cell array, `data`. 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 a Table Record Using the MATLAB® Interface to SQLite

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 6-22
- “Export Multiple Records from the MATLAB Workspace” on page 6-27
- “Export Data Using Bulk Insert” on page 6-31
- “Import Data Using the MATLAB® Interface to SQLite” on page 6-75

- "Roll Back Data After Updating a Record" on page 6-19

Input Arguments

`conn` — Database connection

database connection object | SQLite connection object

Database connection, specified as a database connection object or SQLite connection object created using `database` or `sqlite`.

`tablename` — Database table name

character vector

Database table name, specified as a character vector denoting the name of a table in your database.

Data Types: `char`

`colnames` — Database table column names

cell array of character vectors

Database table column names, specified as a cell array of one or more character vectors to denote the columns in the existing database table `tablename`.

Example: `{'col1','col2','col3'}`

Data Types: `cell`

`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 connection object `conn`.

For a database connection object, you do not specify the type of data that you are exporting. The data is exported in its current MATLAB format. If `data` is a structure, field names in the structure must match `colnames`. If `data` is a table or a dataset array, the variable names in the table or dataset array must match `colnames`. If `data` is a structure, table, or dataset array, 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

To insert dates and timestamps with the native ODBC interface, use the format `'YYYY-MM-DD HH:MM:SS.MS'`.

For a SQLite connection object, the dataset array is not supported. Only `DOUBLE`, `INT64`, and `CHAR` data types are supported.

More About

- "Inserting Data Using the Command Line" on page 2-197
- "Connecting to a Database Using the Native ODBC Interface" on page 3-18
- "Working with the MATLAB Interface to SQLite" on page 2-6

See Also

`close` | `commit` | `database` | `fastinsert` | `rollback`

Introduced before R2006a

isconnection

(Not recommended) Determine if database connections are valid

Compatibility

`isconnection` will be removed in a future release. Use `isopen` instead.

Syntax

```
a = isconnection(conn)
```

Description

`a = isconnection(conn)` returns `1` if the database connection `conn` is valid, or returns `0` otherwise.

Examples

Check if the database connection `conn` is valid.

```
a = isconnection(conn)
a =
     1
```

See Also

`database` | `isreadonly` | `ping`

Introduced before R2006a

isdriver

Detect whether driver is valid JDBC driver object

Compatibility

`isdriver` has been removed.

Syntax

```
a = isdriver(d)
```

Description

`a = isdriver(d)` returns `1` if `d` is a valid JDBC driver object. It returns `0` otherwise.

Examples

Check if `d` is a valid JDBC driver object.

```
a = isdriver(d)
a =
     1
```

See Also

`get`

Introduced before R2006a

isjdbc

Detect whether driver is JDBC compliant

Compatibility

`isjdbc` has been removed.

Syntax

`a = isjdbc(d)`

Description

`a = isjdbc(d)` returns `1` if the driver object `d` is JDBC compliant. It returns `0` otherwise.

Examples

Verify whether the database driver object `d` is JDBC compliant.

```
a = isjdbc(d)
a =
     1
```

See Also

`get`

Introduced before R2006a

isnullcolumn

(Not recommended) Determine if last record read in resultset is `NULL`

Compatibility

`isnullcolumn` has been removed.

Syntax

```
a = isnullcolumn(rset)
```

Description

`a = isnullcolumn(rset)` returns `1` if the last record read in the resultset `rset` is `NULL`. It returns `0` otherwise.

Examples

Example 1 — Result Is Not NULL

`isnullcolumn` returns not null.

1 Run:

```
curs = fetch(curs,1);
rset = resultset(curs);
isnullcolumn(rset)
ans =
     0
```

2 Verify this result.

```
curs.Data
ans =
    [1400]
```

Example 2 — Result Is NULL

`isnullcolumn` returns null.

1 Run:

```
curs = fetch(curs,1);
rset = resultset(curs);
isnullcolumn(rset)
ans =
     1
```

2 Verify this result.

```
curs.Data
ans =
    [NaN]
```

See Also

`fetch` | `get` | `resultset`

Introduced before R2006a

isreadonly

Determine if database connection is read only

Syntax

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

Description

`a = isreadonly(conn)` returns `1` if the database connection `conn` is read only. It returns `0` otherwise.

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.

More About

- "Connecting to a Database" on page 2-191
- "Connecting to a Database Using the Native ODBC Interface" on page 3-18

See Also

`database` | `isopen`

Introduced before R2006a

isurl

Detect whether database URL is valid

Compatibility

`isurl` has been removed.

Syntax

```
a = isurl(d, 's')
```

Description

`a = isurl(d, 's')` returns `1` if the database URL `s` for the driver object `d` is valid. It returns `0` otherwise.

The URL `s` is of the form `jdbc:odbc:`*`name`* or *`name`*.

Examples

Check whether the database URL `jdbc:odbc:thin:@144.212.123.24:1822:` is valid for driver object `d`.

```
a = isurl(d, 'jdbc:odbc:thin:@144.212.123.24:1822:')
a =
     1
```

This indicates that the database URL is valid for `d`.

See Also

`get`

Introduced before R2006a

logintimeout

Set or get time allowed to establish database connection

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 may freeze.

Note: Apple Mac OS 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
```

More About

- “Connecting to a Database” on page 2-191
- “Connecting to a Database Using the Native ODBC Interface” on page 3-18

See Also

`database` | `get` | `isopen` | `isreadonly` | `set`

Introduced before R2006a

namecolumn

Map resultset column name to resultset column index

Compatibility

`namecolumn` will be removed in a future release.

Syntax

`x = namecolumn(rset,n)`

Description

`x = namecolumn(rset,n)` maps a resultset column name `n` to its resultset column index. `rset` is the resultset and `n` is a character vector or cell array of character vectors containing the column names.

Examples

1 Get the indices for the column names `DNAME` and `LOC` resultset object `rset`.

```
x  =  namecolumn(rset, {'DNAME';'LOC'})
x =
     2     3
```

The results show that `DNAME` is column `2` and `LOC` is column `3`.

2 Get the index only for the `LOC` column.

```
x  =  namecolumn(rset, 'LOC')
```

See Also

`columnnames` | `resultset`

Introduced before R2006a

ping

Retrieve status information about database connection

Syntax

```
ping(conn)
```

Description

`ping(conn)` retrieves the status of the database connection `conn`.

Examples

Retrieve Status of an ODBC Connection

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

```
conn = database(dbname,username,pwd);
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Retrieve the status of the Oracle connection.

```
ping(conn)
```

```
ans = 

       DatabaseProductName: 'Oracle'
    DatabaseProductVersion: '11.02.0010'
            JDBCDriverName: 'JDBC-ODBC Bridge (SQORA32.DLL)'
         JDBCDriverVersion: '2.0001 (11.02.0001)'
    MaxDatabaseConnections: 0
           CurrentUserName: 'username'
               DatabaseURL: 'jdbc:odbc:dbname'
```

```
    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 allows automatic commit of transactions.

Close the connection.

```
close(conn)
```

Retrieve Status of a 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 allows automatic commit of transactions.

Close the connection.

```
close(conn)
```


- "Import Data from Databases into MATLAB" on page 6-4
- "Export Data to New Record in Database" on page 6-22

Input Arguments

`conn` — Database connection
database connection object

Database connection, specified as a database connection object created using `database`.

More About

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.

- "Connecting to a Database" on page 2-191

- "Connecting to a Database Using the Native ODBC Interface" on page 3-18

See Also

`close` | `database` | `dmd` | `get` | `isopen` | `set` | `supports`

Introduced before R2006a

primarykeys

Get primary key information for database table or schema

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
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

Introduced before R2006a

procedurecolumns

Get stored procedure parameters and result columns of catalogs

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`

Introduced before R2006a

procedures

Get stored procedures for catalogs

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 `conn`, and fetch all data:

```
curs = exec(conn,'sp_customer_list');
curs = fetch(curs)
curs =

  cursor with properties:

     Attributes: []
           Data: {10x2 cell}
 DatabaseObject: [1x1 database]
       RowLimit: 0
       SQLQuery: 'sp_customer_list'
        Message: []
           Type: 'Database Cursor Object'
      ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
         Cursor: ...
   [1x1 com.mathworks.toolbox.database.sqlExec]
      Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
          Fetch: ...
   [1x1 com.mathworks.toolbox.database.fetchTheData]
```

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

Introduced before R2006a

querybuilder

(Not recommended) Start Visual Query Builder GUI to import and export data

Compatibility

The `querybuilder` function will be removed in a future release. Use Database Explorer instead.

Syntax

```
querybuilder
```

Description

`querybuilder` starts Visual Query Builder (VQB), which is the Database Toolbox GUI.

Tip To populate the VQB **Schema** and **Catalog** fields, you must associate your user name with schemas or catalogs before starting VQB.

Examples

For details about Visual Query Builder, including examples, see the VQB **Help** menu or "Getting Started with Visual Query Builder" on page 5-2.

Introduced before R2006a

querytimeout

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 `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 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`

Introduced before R2006a

register

Load database driver

Compatibility

`register` has been removed.

Syntax

```
register(d)
```

Description

`register(d)` loads the database driver object `d`. Use `unregister` to unload the driver.

Although `database` automatically loads a driver, `register` allows you to use `get` to view properties of the driver before connecting to the database.

Examples

1. `register(d)` loads the database driver object `d`.
2. `get(d)` returns properties of the driver object.

See Also

`get` | `set`

Introduced before R2006a

resultset

Construct resultset object

Compatibility

`resultset` will be removed in a future release.

Syntax

```
rset = resultset(curs)
```

Description

`rset = resultset(curs)` creates a resultset object `rset` for the cursor `curs`. To get properties of `rset`, create a resultset metadata object using `rsmd`, or make calls to `rset` using applications based on Oracle Java.

Run `namecolumn` on `rset`. Use `close` to close the resultset, which frees up resources.

Examples

Construct a resultset object `rset`.

```
rset = resultset(curs)
rset =
     Handle: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
```

See Also

`close` | `exec` | `namecolumn` | `rsmd`

Introduced before R2006a

rollback

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:

   ```
   get(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`.

More About

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`

Introduced before R2006a

rows

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 the Number of Rows in the Cursor

After executing an SQL statement, return the number of rows in the database cursor object generated by `fetch`.

Establish connection `conn` to a MySQL database with user name `username` and password `pwd`.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

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 database 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 6-56

Input Arguments

`curs` — Database cursor

database cursor object

Database cursor, specified as an open SQL database cursor object generated using `fetch`.

Output Arguments

`numrows` — Number of rows in database cursor object

scalar

Number of rows in the database cursor object, returned as a scalar.

See Also

`close` | `cols` | `database` | `exec` | `fetch` | `get` | `rsmd`

Introduced before R2006a

rsmd

Construct resultset metadata object

Compatibility

`rsmd` will be removed in a future release.

Syntax

```
rsmeta = rsmd(rset)
```

Description

`rsmeta = rsmd(rset)` creates a resultset metadata object `rsmeta`, for the resultset object `rset`. Get properties of `rsmeta` using `get` or make calls to `rsmeta` using applications that are based on Oracle Java.

Examples

Create a resultset metadata object `rsmeta`.

```
rsmeta=rsmd(rset)
rsmeta =
     Handle: [1x1 sun.jdbc.odbc.JdbcOdbcResultSetMetaData]
```

Use `v = get(rsmeta)` and `v.property` to view properties of the resultset metadata object.

See Also

`exec` | `get` | `resultset`

Introduced before R2006a

runsqlscript

Run SQL script on database

Syntax

```
results = runsqlscript(conn,sqlfilename)
results = runsqlscript(conn,sqlfilename,Name,Value)
```

Description

`results = runsqlscript(conn,sqlfilename)` runs the SQL commands in the file `sqlfilename` on the connected database, and returns a cursor array.

`results = runsqlscript(conn,sqlfilename,Name,Value)` uses additional options specified by one or more `Name,Value` pairs.

Examples

Run SQL Script

Run SQL commands from a file on a connected data source.

To get the file of SQL commands, navigate to `\toolbox\database\dbdemos\compare_sales.sql` in your MATLAB root folder, or copy and paste the path into your current working folder.

Connect to the Microsoft Access database with the data source name `dbtoolboxdemo` using the native ODBC interface.

```
conn = database.ODBCConnection('dbtoolboxdemo','','');
```

Run the SQL script `compare_sales.sql`.

```
results = runsqlscript(conn,'compare_sales.sql')
```

```
results =

1x2 array of cursor objects
```

The SQL script has two queries, and returns two results when executed.

Display the results for the second query.

```
results(2)
```

```
ans =

  cursor with properties:

        Attributes: []
              Data: {4x6 cell}
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: [1x309 char]
           Message: ''
              Type: 'Database Cursor Object'
         ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
            Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
         Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
             Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

Display the `resultset` returned for the second query.

```
results(2).Data
```

```
ans =

    'Painting Set'    'Terrific Toys'         'London'    [3000]  [2400]  [1800]
    'Victorian Doll'  'Wacky Widgets'         'Adelaide'  [1400]  [1100]  [ 981]
    'Sail Boat'       'Incredible Machines'   'Dublin'    [3000]  [2400]  [1500]
    'Slinky'          'Doll's Galore'         'London'    [3000]  [1500]  [1000]
```

Get the column names for the data returned by the second query.

```
names = columnnames(results(2))
```

```
names =

'productDescription','supplierName','city','Jan_Sales','Feb_Sales','Mar_Sales'
```

Close the cursor array and connection.

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

Run SQL Script in Row Increments

Run SQL commands from a file on a connected data source in two-row increments.

To get the file of SQL commands, navigate to `\toolbox\database\dbdemos\compare_sales.sql` in your MATLAB root folder, or copy and paste the path into your current working folder.

Connect to the Microsoft Access database with the data source name `dbtoolboxdemo` using the native ODBC interface.

```
conn = database.ODBCConnection('dbtoolboxdemo','','');
```

Run the SQL script `compare_sales.sql` and specify two row increments.

```
results = runsqlscript(conn,'compare_sales.sql','rowInc',2)
```

```
results =

1x2 array of cursor objects
```

The SQL script has two queries, and returns two results when executed.

Display the resultset returned for the second query.

```
results(2).Data
```

```
ans =

    'Painting Set'      'Terrific Toys'    'London'      [3000]    [2400]    [1800]
    'Victorian Doll'    'Wacky Widgets'    'Adelaide'    [1400]    [1100]    [ 981]
```

Only the first two rows of the results are returned.

Fetch the next increment of two rows.

```
res2 = fetch(results(2),2);
res2.Data
```

```
ans =

    'Sail Boat'    'Incredible Machines'    'Dublin'    [3000]    [2400]    [1500]
    'Slinky'       'Doll's Galore'          'London'    [3000]    [1500]    [1000]
```

Close the cursor arrays and connection.

```
close(results)
close(res2)
close(conn)
```

Run SQL Script to Fetch Data in Batches

Run SQL commands from a file on a connected data source with automated batching. Use this method to avoid Java heap memory issues when the SQL script returns a large amount of data.

To get the file of SQL commands, navigate to `\toolbox\database\dbdemos\compare_sales.sql` in your MATLAB root folder, or copy and paste the path into your current working folder.

Connect to the Microsoft Access database with the data source name `dbtoolboxdemo` using the native ODBC interface.

```
conn = database.ODBCConnection('dbtoolboxdemo','','');
```

Turn on batching for `fetch`.

```
setdbprefs('FetchInBatches', 'yes')
```

Set appropriate batch size depending on the size of the resultset you expect to fetch. For example, if you expect about a 100,000 rows in the output, a batch size of 10,000 is a good starting point. The larger the `FetchBatchSize` value, the fewer trips between Java and MATLAB, and the memory consumption is greater for each batch. There are several factors that determine the optimal value for `FetchBatchSize`. These factors are some examples:

- Size per row being retrieved
- Java heap memory value
- Default fetch size of the driver
- System architecture

Hence, the `FetchBatchSize` might vary from site to site. For details about estimating a value for `FetchBatchSize`, see “Preference Settings for Large Data Import” on page 5-19.

```
setdbprefs('FetchBatchSize', '2')
```

Run the SQL script `compare_sales.sql`.

```
results = runsqlscript(conn, 'compare_sales.sql')
```

```
results =

1x2 array of cursor objects
```

Batching occurs internally within `fetch`, in that it fetches in increments of two rows at a time. The batching preferences are applied to all the queries in the SQL script.

- “Import Data from Databases into MATLAB” on page 6-4

Input Arguments

`conn` — Database connection
database connection object

Database connection, specified as a database connection object created using `database`.

`sqlfilename` — File name of SQL commands
character vector

File name of SQL commands to run, specified as a character vector. 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`

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: `'RowInc',3,'QTimeOut',60` specifies that results be returned in increments of three rows and the query time out in 60 seconds

`'rowInc'` — Row limit
`0` implies all rows (default) | positive 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 scalar value. Use `rowInc` when importing large amounts of data. Retrieving data in increments helps reduce overall retrieval time.

Example: `'rowInc',5`

Data Types: `double`

`'QTimeOut'` — Query timeout
`0` implies unlimited time (default) | positive scalar

Query timeout (in seconds), specified as the comma-separated pair consisting of `'QTimeOut'` and a positive scalar value.

Example: `'QTimeOut',180`

Data Types: `double`

Output Arguments

`results` — Query results
cursor array

Query results from executing the SQL commands, returned as a cursor array. The number of elements in `results` is equal to the number of batches in the file `sqlfilename`.

`results(M)` contains the results from executing the `M`th SQL batch in the SQL script. If the batch returns a `resultset`, 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. `runsqlscript` is not designed to handle SQL scripts containing continuous PL/SQL blocks with `BEGIN` and `END`, such as stored procedure definitions or trigger definitions. However, table definitions do work.
- An SQL script containing any of the following can produce unexpected results:
  - Apostrophes that are not escaped, including the ones in comments. For example, write the character vector `'Here's the code'` as `'Here''s the code'`.
  - Nested comments.
- An SQL script containing more than 25,000 characters causes `runsqlscript` to return an error.

More About

Batch

One or more SQL statements terminated by either a semicolon or the keyword `GO`.

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 `resultsets`.
- You can set preferences for the `resultsets` using `setdbprefs`. Preference settings apply to all queries in the SQL script. For example, if the `DataReturnFormat` is set to numeric, all the `resultsets` return as numeric matrices.

- “Configuring a Driver and Data Source” on page 2-16
- “Generate SQL and MATLAB Code” on page 4-25
- “Selecting Data” on page 2-195
- “Preference Settings for Large Data Import” on page 5-19

See Also

`close` | `database` | `fetch` | `resultset` | `setdbprefs`

Introduced in R2012a

runstoredprocedure

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 or the ODBC/JDBC bridge. For details, see "Connecting to a Database" on page 2-191. If you are using the native ODBC interface to connect to your database, use `exec` to call the stored procedure.

Syntax

```
results = runstoredprocedure(conn,spname)
results = runstoredprocedure(conn,spname,inputargs)
results = runstoredprocedure(conn,spname,inputargs,outputtypes)
```

Description

`results = runstoredprocedure(conn,spname)` calls the stored procedure `spname` 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,spname,inputargs)` calls the stored procedure that accepts one or more input arguments `inputargs`.

`results = runstoredprocedure(conn,spname,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 a Database” on page 2-191. 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 a Database” on page 2-191. 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 a Database" on page 2-191. 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 a Database” on page 2-191. 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 a Database” on page 2-191. 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 a Stored Procedure That Returns Data" on page 6-44

Input Arguments

`conn` — Database connection
database connection object

Database connection, specified as a database connection object created using `database`.

`spname` — Stored procedure name
character vector

Stored procedure name, specified as a character vector that contains the name of the stored procedure that is defined and instantiated in your database.

Data Types: `char`

`inputargs` — Input arguments
cell array

Input arguments, specified as a cell array of one or more values for each input argument of the stored procedure. Input arguments can be only basic data types such as double, character vector, logical, and so on.

Data Types: `cell`

`outputtypes` — Output types
cell array

Output types, specified as a cell array of one or more Java data types for the output arguments of the stored procedure. Some JDBC drivers do not support all `java.sql.Types`. Consult your JDBC driver documentation to find the supported types. Match them to the data types found in your stored procedure.

Example: `{java.sql.Types.NUMERIC}`

Data Types: `cell`

Output Arguments

`results` — Stored procedure results

logical | cell array

Stored procedure results, returned as a logical or cell array.

`runstoredprocedure` returns a logical `1` when calling the stored procedure returns a data set. Otherwise, `runstoredprocedure` returns a logical `0`. If the stored procedure returns a data set, use `exec` and `fetch` to call the stored procedure and retrieve the data set. For details, see “Call a Stored Procedure That Returns Data” on page 6-44.

`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.

More About

- “Connecting to a Database” on page 2-191

See Also

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

Introduced in R2006b

schemas

Get database schema names

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 6-39

Input Arguments

`conn` — Database connection
database connection object

Database connection, specified as a database connection object created using `database`.

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`

Introduced in R2010a

set

Set properties for database or cursor object

Compatibility

`drivermanager` 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`.

Allowable values for `object` are:

- "Database Connection Objects" on page 8-237, created using `database`
- "Cursor Objects" on page 8-237, created using `exec` or `fetch`

You cannot set all of these properties for all databases. You receive an error message when you try to set a property that the database does not support.

For database 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`.

Database Connection Objects

The allowable values for *`property`* and `value` for a database connection object appear in the following table.

Property	Value	Description
`'AutoCommit'`	`'on'`	The software writes and automatically commits database data when you run `datainsert`, `fastinsert`, `insert`, or `update`. You cannot use `rollback` to reverse this process.
	`'off'`	The software does not automatically commit database data when you run `datainsert`, `fastinsert`, `insert`, or `update`. Use `rollback` to reverse this process. When you are sure that your data is correct, use the `commit` function to commit it to the database. Alternatively, use `exec` to roll back or commit data to the database.
`'ReadOnly'`	`0`	Not read only; that is, writable
	`1`	Read only
`'TransactionIsolation'`	positive integer	Current transaction isolation level

Note: For some databases, if you insert data and then close the database connection without committing the data to the database, the data gets committed automatically. Your database administrator can tell you whether your database behaves this way.

Cursor Objects

The allowable *`property`* and `value` for a cursor object appear in the following table.

Property	Value	Description
`'RowLimit'`	positive integer	Sets the `RowLimit` for `fetch`. Specify this property instead of passing `RowLimit` as

Property	Value	Description
		an argument to the `fetch` function. When you define `RowLimit` for `fetch` by using `set`, `fetch` behaves differently depending on what type of database you are using.

Examples

Example 1 — Set RowLimit for Cursor

This example does the following:

- Establishes a JDBC connection to a data source.
- Runs `fetch` to retrieve data from the table `EMP`.
- Sets `RowLimit` 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.

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Example 2 — Set the AutoCommit Flag to On

This example shows what happens when you 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

This example shows the results of running `datainsert` and `commit` to 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

This example runs `update` to update data in a database whose `AutoCommit` flag is `off`. It then uses `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`

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 for database preferences to the structure `v`.

`setdbprefs(preference)` returns the current value for the specified `preference`.

`setdbprefs(preference,value)` sets the specified `preference` to `value`. Once 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
```

```
           DataReturnFormat: 'cellarray'
              ErrorHandling: 'store'
             NullNumberRead: '0'
```

```
          NullNumberWrite: 'NaN'
           NullStringRead: 'null'
          NullStringWrite: 'null'
       JDBCDataSourceFile: 'C:\hold_x\jdbcConfig_test.mat'
    UseRegistryForSources: 'yes'
 TempDirForRegistryOutput: 'C:\Work'
       DefaultRowPreFetch: '10000'
           FetchInBatches: 'no'
           FetchBatchSize: '1000'
```

Change a Preference

Set a database preference to another value.

Display the current value of the `NullNumberRead` database preference.

```
setdbprefs('NullNumberRead')
```

```
NullNumberRead: '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 the DataReturnFormat Preference

Changing the database preference `DataReturnFormat` affects the way data is returned to the MATLAB workspace.

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');
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see the `database` function.

Import data into the MATLAB workspace.

```
curs = exec(conn,...
'select productnumber,productdescription from producttable');
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.

```
curs = exec(conn,...
'select productnumber,productdescription from producttable');
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`. Therefore, the `productDescription` values cannot be read when they are imported into the MATLAB workspace using the `numeric` format. Therefore, MATLAB treats them 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.

```
curs = exec(conn,...
```

```
'select productnumber,productdescription from producttable');
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 the Write Format for NULL Numbers

Changing the write format for `NULL` numbers allows the insertion of a `NaN` as a `NULL` in the database.

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`.

```
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...'
```

Specify data `ex_data` to export into `inventoryTable`. `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.

```
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...'
    [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

Changing the error handling database preferences affects the display of errors in MATLAB.

Specify the `store` format for the `ErrorHandling` preference.

```
setdbprefs('ErrorHandling','store')
```

With the `ErrorHandling` preference setting set to `store`, errors generated by running `database` or `exec` are stored in the `Message` field 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 `curs` contains the executed query. Close the cursor object. Fetch data from a closed cursor object.

```
curs = exec(conn,'select productdescription from producttable');
close(curs)
curs = fetch(curs,3)
```

```
curs =

  cursor with properties:

        Attributes: []
              Data: 0
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: 'select productdescription from producttable'
           Message: 'Invalid fetch cursor.'
              Type: 'Database Cursor Object'
         ResultSet: 0
            Cursor: 0
         Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
             Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

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.

```
curs = exec(conn,'select productdescription from producttable');
close(curs)
curs = fetch(curs,3);
```

```
Error using cursor/fetch>errorhandling (line 491)
Invalid fetch cursor.

Error in cursor/fetch (line 460)
            errorhandling(outCursor.Message);
```

The error generated by this operation appears immediately in the Command Window.

Specify the `empty` format for the `ErrorHandling` preference.

```
setdbprefs('ErrorHandling','empty')
```

With the `ErrorHandling` preference setting set to `empty`, errors generated while running `database` or `exec` are stored in the `Message` field of the returned connection or cursor object. In addition, objects that cannot be created are returned as empty handles, `[]`.

Fetch data from a cursor from an invalid table `invalidTable`.

```
curs = exec(conn, 'select * from invalidTable')
curs = fetch(curs)
```

```
curs =

  cursor with properties:

        Attributes: []
              Data: []
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: 'select * from invalidTable'
```

```
      Message: [1x102 char]
         Type: 'Database Cursor Object'
    ResultSet: 0
       Cursor: 0
    Statement: 0
        Fetch: 0
```

The error appears in the cursor object `Message` field. Furthermore, the `Data` field contains empty handles because no attributes could be created. If the `ErrorHandling` preference setting is set to `store`, the `Data` field contains `0`.

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 a Structure” on page 8-249.

Assign Values to a 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.TempDirForRegistryOutput = 'C:\Work'

s =
            DataReturnFormat: 'numeric'
              NullNumberRead: '0'
    TempDirForRegistryOutput: 'C:\Work'
```

Set preferences using the values in s.

```
setdbprefs(s)
```

Run `setdbprefs` to check your preferences settings.

```
setdbprefs
```

```
            DataReturnFormat: 'numeric'
               ErrorHandling: 'store'
              NullNumberRead: '0'
             NullNumberWrite: 'NaN'
              NullStringRead: 'null'
             NullStringWrite: 'null'
          JDBCDataSourceFile: ''
       UseRegistryForSources: 'yes'
    TempDirForRegistryOutput: 'C:\Work'
          DefaultRowPreFetch: '10000'
              FetchInBatches: 'no'
              FetchBatchSize: '1000'
```

Return Values to a Structure

Capture all preferences and their values in a structure.

Assign values for all preferences to s.

```
s = setdbprefs
```

```
s =

            DataReturnFormat: 'cellarray'
               ErrorHandling: 'store'
              NullNumberRead: 'NaN'
             NullNumberWrite: 'NaN'
              NullStringRead: 'null'
             NullStringWrite: 'null'
          JDBCDataSourceFile: ''
       UseRegistryForSources: 'yes'
    TempDirForRegistryOutput: 'C:\Work'
          DefaultRowPreFetch: '10000'
              FetchInBatches: 'no'
              FetchBatchSize: '1000'
```

Use the MATLAB tab completion feature when obtaining the value for a preference.

```
s.U
```

Press the **Tab** key, and then **Enter**. MATLAB completes the field and displays the value.

```
s.UseRegistryForSources

ans =

    yes
```

Save Preferences

You can save your preferences to a MAT-file to use them in future MATLAB sessions.

Suppose you want to reuse preferences that you set for fetching large data. Assign the preferences to the variable `FetchLargeData` and save them to a MAT-file `FetchLargeDataPrefs` in your current folder.

```
FetchLargeData = setdbprefs;
save FetchLargeDataPrefs.mat FetchLargeData
```

Later, load the data and restore the preferences.

```
load FetchLargeDataPrefs.mat
setdbprefs(FetchLargeData)
```

- "Import Data from Databases into MATLAB" on page 6-4

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 these preferences. 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')`.

Allowable Values	Description
`'cellarray'` (default)	Import nonnumeric data into MATLAB cell arrays.
`'table'`	Import data into a MATLAB `table` object. 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 option 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 these preferences. Set this parameter before you run `exec`. For example, to specify storing errors in the `Message` field of the returned connection object, enter `setdbprefs('ErrorHandling','store')`.

Allowable Values	Description
`'store'` (default)	Store errors from running `database` in the `Message` field of the returned connection object. Store errors from running `exec` in the `Message` field of the returned cursor object.
`'report'`	Immediately display errors from running `database` or `exec` in the Command Window.

Allowable Values	Description
'empty'	Store errors from running `database` in the `Message` field of the returned connection object. Store errors from running `exec` in the `Message` field of the returned cursor object. Objects that cannot be created are returned as empty handles (`[ ]`).

- For importing or exporting data, you can specify how to handle `NULL` data in the MATLAB workspace or the database using these preferences. For example, to import data and display all `NULL` numbers in the database as a `0` in the MATLAB workspace, enter `setdbprefs('NullNumberRead','0')`.

Database Preference	Allowable Values	Description
'NullNumberRead'	Character vector, for example, `'0'`	How `NULL` numbers appear after being imported from a database into the MATLAB workspace. `NaN` is the default value. If `'DataReturnFormat'` is set to `'numeric'`, values such as `'NULL'` cannot be set. If `'DataReturnFormat'` is set to `'cellarray'`, numbers appear as `NaN` and not as empty brackets. Set this parameter before running `fetch`.
'NullNumberWrite'	Character vector, for example, `'NaN'` (default)	How numbers appear in the database after being exported from the MATLAB workspace to a database. Regardless of the value of `'NullNumberWrite'`, a `NULL` value is always written to the database

Database Preference	Allowable Values	Description
		when you input `[]` or `NaN` for a numeric data type.
`'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 `fetch`.
`'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'`.

- `'JDBCDataSourceFile'` — The path to the MAT-file containing JDBC data sources. Specify this file path as a character vector. For example, to specify the file path as `'D:/file.mat'`, enter `setdbprefs('JDBCDataSourceFile','D:/file.mat')`.
- When importing data, you can use these preferences for controlling 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')`.

Database Preference	Allowable Values	Description
`'DefaultRowPreFetch'`	Input numeric value, default value is `'10000'`	Number of rows fetched from the database server at a time for any query. Higher numbers result in fewer trips to the server.

Database Preference	Allowable Values	Description
		Caution This property is applicable only for databases that allow setting this number, such as Oracle.
`'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'`, `fetch` and `runsqlscript` import the data in batches in size of `'FetchBatchSize'`. For an example, see `fetch`.
`'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'`, `fetch` and `runsqlscript` import the data in batches in size of `'FetchBatchSize'`. For an example, see `fetch`. For details about estimating a `'FetchBatchSize'` value, see "Preference Settings for Large Data Import" on page 5-19.

- To manage ODBC data sources using VQB, use these preferences. For example, to specify the folder where VQB writes ODBC registry settings as `'D:/work'`, enter `setdbprefs('TempDirForRegistryOutput','D:/work')`.

Database Preference	Allowable Values	Description
`'UseRegistryForSources`	`'yes'` (default) or `'no'`	When set to `yes`, VQB searches the Microsoft Windows registry for ODBC data sources that are not found in the system `ODBC.INI` file. This message can appear: Registry editing has been disabled by your administrator. You can ignore this harmless message.
`'TempDirForRegistryOut`	Character vector that specifies a file path, for example, `'D:/work'`	Folder where VQB writes `ODBC` registry settings when you run `getdatasources`. Use this preference when you add data sources and do not have write access to the MATLAB current folder. The default is the Windows temporary folder. See the folder by entering this command: `getenv('temp')`. If you specify a folder to which you do not have write access or which does not exist, this error appears: Cannot export*`folder-name`*\ODBC.INI: Error opening the file. There may be a disk or file system error.

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 you specify.

Data Types: `struct`

Output Arguments

`v` — Database preferences
structure

Database preferences, returned as a structure containing database preference settings and values.

More About

Tips

- For a visual way to set database preferences, select **Preferences**. Click **Database Toolbox**. Enter values for each database preference.

- “Preference Settings for Large Data Import” on page 5-19
- “Working with Preferences” on page 5-15

See Also

`clear` | `close` | `database` | `exec` | `fastinsert` | `fetch` | `getdatasources`

Introduced before R2006a

sqlite

Create SQLite connection

The `sqlite` function creates a SQLite connection object. You can use this object to connect to a SQLite database file using the MATLAB interface to SQLite. The MATLAB interface to SQLite lets you work with SQLite database files without installing and administering a database or driver. For details, see "Working with the MATLAB Interface to SQLite" on page 2-6.

A SQLite connection object is one of the two available database connection types. The other creates a database connection object using the function `database`. This object lets you connect to various databases using different drivers that you install and administer. For details, see "Connecting to a Database" on page 2-191.

Syntax

```
conn = sqlite(dbfile)
conn = sqlite(dbfile,mode)
```

Description

`conn = sqlite(dbfile)` connects to an existing SQLite database file `dbfile`.

`conn = sqlite(dbfile,mode)` connects to an existing SQLite database file or creates a database file depending on the mode type `mode`.

Examples

Create SQLite Connection to an Existing Database File

Create a SQLite connection `conn` to the MATLAB® interface to SQLite using an existing database file `tutorial.db`. Specify the file name in the current working folder.

```
dbfile = fullfile(pwd,'tutorial.db');
```

```
conn = sqlite(dbfile)
```

```
conn = 

  sqlite with properties:

      Database: 'C:\TEMP\Bdoc16b_437511_11332\IB_CPU_9\tp38a6bb90_3a86_4a0...'
        IsOpen: 1
    IsReadOnly: 0
```

`conn` is a SQLite connection object with these properties:

- SQLite database file name.
- SQLite connection is open.
- SQLite connection is writable.

Close the SQLite connection.

```
close(conn)
```

Create SQLite Connection Using a New Database File

Create a SQLite connection `conn` to the MATLAB® interface to SQLite using a new database file `tutorial.db`. Specify the file name in the current working folder.

```
dbfile = fullfile(pwd,'tutorial.db');

conn = sqlite(dbfile,'create')
```

```
conn = 

  sqlite with properties:

      Database: 'C:\TEMP\Bdoc16b_437511_11332\IB_CPU_9\tp38a6bb90_3a86_4a0...'
        IsOpen: 1
    IsReadOnly: 0
```

`conn` is a SQLite connection object with these properties:

- SQLite database file name.
- SQLite connection is open.

- SQLite connection is writable.

Close the SQLite connection.

```
close(conn)
```

Create a Read-Only SQLite Connection

Create a read-only SQLite connection `conn` to the MATLAB® interface to SQLite using an existing database file `tutorial.db`. Specify the file name in the current working folder.

```
dbfile = fullfile(pwd,'tutorial.db');

conn = sqlite(dbfile,'readonly')
```

```
conn = 

  sqlite with properties:

      Database: 'C:\TEMP\Bdoc16b_437511_11332\IB_CPU_9\tp38a6bb90_3a86_4a0...'
        IsOpen: 1
    IsReadOnly: 1
```

`conn` is a SQLite connection object with these properties:

- SQLite database file name.
- SQLite connection is open.
- SQLite connection is read only.

Close the SQLite connection.

```
close(conn)
```

- “Import Data Using the MATLAB® Interface to SQLite” on page 6-75

Input Arguments

`dbfile` — SQLite database file

character vector

SQLite database file, specified as a character vector. You can use the database file to store data and import and export it to MATLAB.

Data Types: `char`

`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.

Output Arguments

`conn` — SQLite connection
SQLite connection object

SQLite connection, returned as a SQLite connection object. This object has these properties.

Property	Description
`Database`	SQLite database file name
`IsOpen`	Logical `1` or `0` that denotes whether the database connection is open. `1` means that the database connection is open, and `0` means that it is closed or invalid.
`IsReadOnly`	Logical `1` or `0` that denotes whether the database connection is read only. `1` means that the database connection is read only, and `0` means that it is writable.

More About

Tips

- To use all the Database Toolbox 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 2-83
  - “SQLite JDBC for Mac OS X” on page 2-160
  - “SQLite JDBC for Linux” on page 2-167

- “Working with a Database and MATLAB” on page 2-3
- “Working with the MATLAB Interface to SQLite” on page 2-6
- “Configuring a Driver and Data Source” on page 2-16

See Also

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

Introduced in R2016a

sql2native

Convert JDBC SQL grammar to SQL grammar native to system

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`.

More About

- “Creating and Running SQL Queries” on page 1-9

See Also

`database`

Introduced before R2006a

supports

Detect whether property is supported by database metadata object

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

Introduced before R2006a

tableprivileges

Return database table privileges

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 database 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 database 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 database 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` | `tables`

Introduced before R2006a

tables

Return database table names

Compatibility

The syntax `tables(conn)` has been removed. Use syntaxes with at least two input arguments 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`.

`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 the Table List for a Catalog Using the Database Connection

Create a database connection `conn` using the native ODBC interface to the Microsoft SQL Server database. For example, the following code assumes that you are connecting to a data source named `MS SQL Server` with user name `username` and password `pwd`.

```
conn = database.ODBCConnection('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 the Table List for a Catalog and Schema Using the Database Connection

Create a database connection `conn` using the native ODBC interface to the Microsoft SQL Server database. For example, the following code assumes that you are connecting to a data source named `MS SQL Server` with user name `username` and password `pwd`.

```
conn = database.ODBCConnection('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 the Table List for a Catalog Using the 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 the Table List for a Catalog and Schema Using the 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 6-39
- “Import Data from Databases into MATLAB” on page 6-4
- “Export Data to New Record in Database” on page 6-22

Input Arguments

`conn` — Database connection

database connection object

Database connection, specified as a database connection object created using `database`.

`dbmeta` — Database metadata

database metadata object

Database metadata, specified as a database metadata object created using `dmd`. To use this object, connect to the database using the JDBC driver or JDBC/ODBC bridge. For details about database connections, see `database`.

`catalog` — Database catalog name

character vector

Database catalog name, specified as a character vector.

Data Types: `char`

`schema` — Database schema name

character vector

Database schema name, specified as a character vector.

Data Types: `char`

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.

More About

- “Connecting to a Database” on page 2-191
- “Connecting to a Database Using the Native ODBC Interface” on page 3-18

See Also

`catalogs` | `close` | `database` | `dmd` | `get` | `schemas`

Introduced in R2010a

unregister

Unload database driver

Compatibility

`unregister` has been removed.

Syntax

```
unregister(d)
```

Description

`unregister(d)` unloads the database driver object `d`, freeing up system resources. If you do not unload a registered driver, it automatically unloads when you end your MATLAB session.

Examples

`unregister(d)` unloads the database driver object `d`.

Introduced before R2006a

update

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`. Existing records in the database table are replaced as specified by the SQL `whereclause` command.

Examples

Update an Existing Record Using a Cell Array

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 `dbtoolboxdemo` with blank user name and password. This database contains the table `inventoryTable` that contains these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Import all data from the `inventoryTable` using `conn`. Store the data in a cell array contained in the cursor object property `curs.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 name that you are updating called `Quantity`.

```
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)
```

Fetch 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 an Existing Record Using a Table

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 `dbtoolboxdemo` with blank user name and password. This database contains the table `inventoryTable` that contains these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Import all data from the `inventoryTable` using `conn`. Store the data in a cell array contained in the cursor object property `curs.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 containing the new data `3400`.

```
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 with `productNumber` equal to `1`.

```
tablename = 'inventoryTable';
whereclause = 'where productNumber = 1';

update(conn,tablename,colnames,data,whereclause)
```

Fetch 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

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 `dbtoolboxdemo` with blank user name and password. This database contains the table `inventoryTable` that contains these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Import all data from the `inventoryTable` using `conn`. Store the data in a cell array contained in the cursor object property `curs.Data`. 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.

```
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)
```

Fetch 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

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 `dbtoolboxdemo` with blank user name and password. This database contains the table `inventoryTable` that contains these columns:

- `productNumber`
- `Quantity`
- `Price`
- `inventoryDate`

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For details, see `database`.

Import all data from `inventoryTable` using `conn`. Store the data in a cell array contained in the cursor object property `curs.Data`. 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.

```
% 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)
```

Fetch 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 a Database" on page 6-25
- "Roll Back Data After Updating a Record" on page 6-19
- "Import Data from Databases into MATLAB" on page 6-4

Input Arguments

`conn` — Database connection

database connection object

Database connection, specified as a database connection object created using `database`.

`tablename` — Database table name

character vector

Database table name, specified as a character vector denoting the name of a table in your database.

Data Types: `char`

`colnames` — Database table column names

cell array of character vectors

Database table column names, specified as a cell array of one or more character vectors to denote the columns in the existing database table `tablename`.

Example: `{'col1','col2','col3'}`

Data Types: `cell`

`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 or the JDBC/ODBC bridge, 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 `NaN`s, convert these entries to an empty value `''`.

If `data` is a structure, field names in the structure must match `colnames`. If `data` is a table or a dataset array, the variable names in the table or dataset array must match `colnames`.

`whereclause` — SQL WHERE clause
character vector | cell array

SQL WHERE clause, specified as a character vector for one condition or a cell array of character vectors for multiple conditions.

Example: `'WHERE productTable.productNumber = 1'`

Data Types: `char`

More About

Tips

- The status of the `AutoCommit` flag determines whether `update` automatically commits the data to the database. View the `AutoCommit` flag status for the connection using `get` and change it using `set`. Commit the data by running `commit` or an SQL commit statement using the `exec` function. Roll back the data by running `rollback` or an SQL ROLLBACK statement using the `exec` function.

- To add new rows instead of replacing existing data, use `datainsert`.
- To update multiple records, the number of SQL WHERE clauses in `whereclause` must match the number of records in `data`.
- The order of records in your database is not constant. Use values of column names to identify records.
- This error message can appear 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.
  ```

  In this case, close the table and rerun the `update` function.
- Running an update operation that matches the one that you just ran can cause this error message to appear.

  ```
  ??? Error using ==> database.update
  Error:Commit/Rollback Problems
  ```

- “Connecting to a Database Using the Native ODBC Interface” on page 3-18

See Also

`close` | `commit` | `database` | `datainsert` | `get` | `rollback` | `set`

Introduced before R2006a

versioncolumns

(Not recommended) Automatically update table columns

Compatibility

`versioncolumns` has been removed.

Syntax

```
vl = versioncolumns(dbmeta, 'cata')
vl = versioncolumns(dbmeta, 'cata', 'sch')
vl = versioncolumns(dbmeta, 'cata', 'sch', 'tab')
```

Description

`vl = versioncolumns(dbmeta, 'cata')` returns a list of columns that automatically update when a row value updates in the catalog `cata`, in the database whose database metadata object is `dbmeta` resulting from a database connection object.

`vl = versioncolumns(dbmeta, 'cata', 'sch')` returns a list of all columns that automatically update when a row value updates in the schema `sch`, in the catalog `cata`, for the database whose database metadata object is `dbmeta` resulting from a database connection object.

`vl = versioncolumns(dbmeta, 'cata', 'sch', 'tab')` returns a list of columns that automatically update when a row value updates in the table `tab`, the schema `sch`, in the catalog `cata`, for the database whose database metadata object is `dbmeta` resulting from a database connection object.

Examples

Get a list of which columns automatically update when a row in the table `BONUS` updates, in the schema `SCOTT`, in the catalog `orcl`, for the database metadata object `dbmeta`.

```
vl = versioncolumns(dbmeta,'orcl','SCOTT','BONUS')
vl =
     {}
```

The results are an empty set, indicating that no columns in the database automatically update when a row value updates.

See Also

columns | dmd | get

Introduced before R2006a

width

Return field size of column in fetched data set

Syntax

```
colsize = width(cursor, colnum)
```

Description

`colsize = width(cursor, 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`

Introduced before R2006a

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.

Create Object

Create a `Neo4jConnect` object using `neo4j`.

Properties

`URL` — Neo4j database connection URL

character vector

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

User name for accessing the Neo4j database, specified as a character vector.

Data Types: `char`

Message — Error message
character vector

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 node 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: []
```

Check the `Message` property of the Neo4j® connection object `neo4jconn`.

```
neo4jconn.Message
```

```
ans =

     []
```

The blank `Message` property indicates a successful connection.

`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

The blank `Message` property indicates a successful Neo4j® database connection.

- "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 7-2

More About

- "Working with the MATLAB Interface to Neo4j" on page 7-8
- "Searching Graph Database Using MATLAB Interface to Neo4j" on page 7-10

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.

Create Object

Create a `Neo4jNode` object using `searchNodeByID` or `searchNode`.

Properties

`NodeID` — Node identifier
double

Node identifier for the unique node in the Neo4j database, specified as a double.

Data Types: `double`

`NodeData` — Node data
structure

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

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- 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 node identifier `2` 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 that contains 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 7-2

More About

- "Working with the MATLAB Interface to Neo4j" on page 7-8
- "Searching Graph Database Using MATLAB Interface to Neo4j" on page 7-10

Introduced in R2016b

neo4j

Connect to Neo4j database

The `neo4j` function creates connections to a Neo4j database. For relational database connections, see "Connecting to a Database" on page 2-191.

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: []
```

Check the `Message` property of the Neo4j® connection object `neo4jconn`.

```
neo4jconn.Message
```

```
ans =

     []
```

The blank `Message` property indicates a successful connection.

`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

The blank `Message` property indicates a successful Neo4j® database connection.

- “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 7-2

Input Arguments

`url` — Neo4j database connection URL

character vector

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

User name for accessing the Neo4j database, specified as a character vector. If no database authentication is required, specify an empty character vector.

Data Types: `char`

`password` — Password
character vector

Password for accessing the Neo4j database, specified as a character vector. If no database authentication is required, specify an empty character vector.

Data Types: `char`

Output Arguments

`neo4jconn` — Neo4j database connection
`Neo4jConnect` object

Neo4j database connection, returned as a `Neo4jConnect` object.

More About

- “Working with the MATLAB Interface to Neo4j” on page 7-8
- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 7-10
- “MATLAB Interface to Neo4j Error Messages” on page 7-13

See Also

`neo4j` | `nodeLabels` | `propertyKeys` | `relationTypes`

Introduced in R2016b

nodeLabels

All node labels in Neo4j database

Syntax

`nlabels = nodeLabels(neo4jconn)`

Description

`nlabels = nodeLabels(neo4jconn)` returns all node labels in the Neo4j database using the Neo4j database connection `neo4jconn`. You can retrieve the entire graph or search for a subgraph using the node labels. To search the graph database for relationship types instead, see `relationTypes`.

Examples

Retrieve 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 7-2

Input Arguments

`neo4jconn` — Neo4j database connection
`Neo4jConnect` object

Neo4j database connection, specified as a `Neo4jConnect` object created using 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.

More About

- “Working with the MATLAB Interface to Neo4j” on page 7-8
- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 7-10

- “MATLAB Interface to Neo4j Error Messages” on page 7-13

See Also

`neo4j` | `propertyKeys` | `relationTypes`

Introduced in R2016b

relationTypes

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 7-2

Input Arguments

`neo4jconn` — Neo4j database connection
`Neo4jConnect` object

Neo4j database connection, specified as a `Neo4jConnect` object created using 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.

More About

- "Working with the MATLAB Interface to Neo4j" on page 7-8
- "Searching Graph Database Using MATLAB Interface to Neo4j" on page 7-10

- "MATLAB Interface to Neo4j Error Messages" on page 7-13

See Also

`neo4j` | `nodeLabels` | `propertyKeys`

Introduced in R2016b

propertyKeys

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 `neo4jconn`.

```
propkeys = propertyKeys(neo4jconn)

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 7-2

Input Arguments

`neo4jconn` — Neo4j database connection

`Neo4jConnect` object

Neo4j database connection, specified as a `Neo4jConnect` object created using 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.

More About

- “Working with the MATLAB Interface to Neo4j” on page 7-8
- “MATLAB Interface to Neo4j Error Messages” on page 7-13

See Also

`neo4j` | `nodeLabels` | `relationTypes`

Introduced in R2016b

searchNodeByID

Search for Neo4j database node 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 node identifier `2` 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 that contains 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 7-2

Input Arguments

neo4jconn — Neo4j database connection
`Neo4jConnect` object

Neo4j database connection, specified as a `Neo4jConnect` object created using the function `neo4j`.

`nodeid` — Neo4j database node identifier
numeric scalar

Neo4j database node identifier, specified as a numeric scalar that denotes one specific node in the Neo4j database. If a node identifier is unknown, search for nodes using `searchNode` and 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.

More About

- “Working with the MATLAB Interface to Neo4j” on page 7-8
- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 7-10

See Also
`neo4j` | `nodeDegree` | `nodeRelationTypes`

Introduced in R2016b

searchNode

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'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    1    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    2    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    3    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    4    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    5    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    6    'Person'      [1×1 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 `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`. 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 7-2

Input Arguments

`neo4jconn` — Neo4j database connection
`Neo4jConnect` object

Neo4j database connection, specified as a `Neo4jConnect` object created using the function `neo4j`.

`nlabel` — Neo4j database node label
character vector

Neo4j database node label, specified as a character vector.

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: `nodeinfo = searchNode(neo4jconn,'Person','PropertyKey','name','PropertyValue','User2');`

`'PropertyKey'` — Property key
character vector

Property key, specified as a comma-separated pair consisting of `'PropertyKey'` and a character vector. 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`

`'PropertyValue'` — Property value
character vector

Property value, specified as a comma-separated pair consisting of `'PropertyValue'` and a character vector. 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`

Output Arguments

`nodeinfo` — Node information
`Neo4jNode` object | table

Node information in the Neo4j database that matches the search criteria, returned as a `Neo4jNode` object for one node or a table for multiple nodes.

For multiple nodes, the table contains:

- Row names, which are Neo4j node identifiers of each database node.
- The variable `NodeLabels`, which is a cell array of character vectors that contains the node labels for each database node.
- The variable `NodeData`, which is a cell array of structures that contain node information such as property keys.
- The variable `NodeObject`, which is the `Neo4jNode` object for each database node.

More About

- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 7-10
- “Working with the MATLAB Interface to Neo4j” on page 7-8
- “MATLAB Interface to Neo4j Error Messages” on page 7-13

See Also

`neo4j` | `nodeDegree` | `nodeRelationTypes` | `searchGraph` | `searchNodeByID` | `searchRelation`

Introduced in R2016b

searchRelation

Search relationships for Neo4j database node

Syntax

```
relinfo = searchRelation(neo4jconn,nodeinfo,direction)
relinfo = searchRelation(neo4jconn,nodeinfo,direction,Name,Value)
```

Description

`relinfo = searchRelation(neo4jconn,nodeinfo,direction)` returns relationship information for the origin node `nodeinfo` and relationship direction using the Neo4j database connection `neo4jconn`. The search starts from the origin node. To find an origin node, use `searchNode` or `searchNodeByID`.

`relinfo = searchRelation(neo4jconn,nodeinfo,direction,Name,Value)` returns relationship information with additional options specified by one or more `Name,Value` pair arguments.

Examples

Search Incoming 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.

Retrieve the origin node `nodeinfo` using the Neo4j® database connection `neo4jconn` and node identifier `nodeid`.

```
nodeid = 3;

nodeinfo = searchNodeByID(neo4jconn,nodeid);
```

Search for incoming relationships using the Neo4j® database connection `neo4jconn` and origin node `nodeinfo`.

```
direction = 'in';

relinfo = searchRelation(neo4jconn,nodeinfo,direction)
```

```
relinfo =

  struct with fields:

       Origin: 3
        Nodes: [2×3 table]
    Relations: [1×4 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 starting and ending nodes for each matched relationship.
- `Relations` -- A table containing all matched relationships.

Access the table of nodes.

```
relinfo.Nodes
```

```
ans =

         NodeLabels       NodeData                     NodeObject
         __________    ____________    ______________________________

    1    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    3    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
```

Access the table of relationships.

```
relinfo.Relations
```

```
ans =

         StartNodeID    RelationType    EndNodeID    RelationData
         ___________    ____________    _________    ____________

    3    1              'knows'         3            [1×1 struct]
```

Search Relationships by Type and Distance

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 `neo4jconn` and node identifier `nodeid`.

```
nodeid = 3;

nodeinfo = searchNodeByID(neo4jconn,nodeid);
```

Search for incoming relationships using the Neo4j® database connection `neo4jconn` and origin node `nodeinfo`. Refine the search by filtering for the relationship type `knows` and for nodes at a distance of two or less.

```
direction = 'in';
reltypes = {'knows'};

relinfo = searchRelation(neo4jconn,nodeinfo,direction, ...
    'RelationTypes',reltypes,'Distance',2)
```

```
relinfo = 

  struct with fields:

       Origin: 3
        Nodes: [4×3 table]
    Relations: [3×4 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 starting and ending nodes for each matched relationship.
- `Relations` -- A table containing all matched relationships.

Access the table of nodes.

```
relinfo.Nodes
```

```
ans = 

         NodeLabels      NodeData                  NodeObject
         __________    ____________    ______________________________
```

```
    0    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    1    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    2    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    3    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
```

Access the table of relationships.

```
relinfo.Relations
```

```
ans = 

         StartNodeID    RelationType    EndNodeID    RelationData
         ___________    ____________    _________    ____________

    3    1              'knows'         3            [1×1 struct]
    2    2              'knows'         1            [1×1 struct]
    1    0              'knows'         1            [1×1 struct]
```

- "Explore Graph Database Structure" on page 7-2

Input Arguments

`neo4jconn` — Neo4j database connection

`Neo4jConnect` object

Neo4j database connection, specified as a `Neo4jConnect` object created using 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 one of these values. The relationships are associated with the specified origin node.

Value	Description
'in'	Incoming relationship
'out'	Outgoing relationship

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);`

`'RelationTypes'` — Relationship types

cell array of character vectors

Relationship types, specified as a comma-separated pair consisting of `'RelationTypes'` and a cell array of character vectors. To denote one relationship, specify one character vector in the cell array. To denote multiple relationships, specify multiple character vectors in the cell array.

Example: `'RelationTypes',{'knows'}`

Data Types: `char`

`'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 less than four nodes away from the origin node `nodeinfo`.

Example: `'Distance',3`

Data Types: `double`

Output Arguments

`relinfo` — 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 `nodeinfo`.
`Nodes`	Table that contains node information for each node in the `Relations` table. The `Nodes` table contains: • Row names in the table, which are Neo4j node identifiers of the matched database nodes. • The variable `NodeLabels`, which is a character vector that denotes the node label for each matched database node. • The variable `NodeData`, which is a structure array that contains node information such as property keys for each matched database node. • The variable `NodeObject`, which is the `Neo4jNode` object for each matched database node.
`Relations`	Table that contains relationship information for the nodes in the `Nodes` table. The `Relations` table contains: • Row names in the table, which are Neo4j relationship identifiers. • The variable `StartNodeID`, which is the node identifier for the start node for each matched relationship.

Field	Description
	• The variable `RelationType`, which is a character vector that denotes the relationship type for each matched relationship. • The variable `EndNodeID`, which is the node identifier for the end node for each matched relationship. • The variable `RelationData`, which is a structure array that contains property keys associated with each matched relationship.

More About

- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 7-10
- “Working with the MATLAB Interface to Neo4j” on page 7-8
- “MATLAB Interface to Neo4j Error Messages” on page 7-13

See Also

`neo4j` | `searchGraph` | `searchNode` | `searchNodeByID`

Introduced in R2016b

searchGraph

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 the Neo4j database connection `neo4jconn`. You can search a subgraph or the entire graph.

Examples

Search Graph by Node Labels

Create a Neo4j® database connection using the URL `http://localhost:7474/db/data`, user name `neo4j`, and password `matlab`.

```
url = 'http://localhost:7474/db/data';
username = 'neo4j';
password = 'matlab';

neo4jconn = neo4j(url,username,password);
```

Check the `Message` property of the Neo4j® connection object `neo4jconn`.

```
neo4jconn.Message
```

```
ans =

     []
```

The blank `Message` property indicates a successful connection.

Search the graph for all nodes that are labeled as `'Person'` using the Neo4j® database connection `neo4jconn`.

```
nlabel = {'Person'};

graphinfo = searchGraph(neo4jconn,nlabel)
```

```
graphinfo = 

  struct with fields:

        Nodes: [7×3 table]
    Relations: [8×4 table]
```

`graphinfo` is a structure that contains the results of the search:

- `Nodes` -- A table containing all starting and ending nodes that denote each matched relationship.
- `Relations` -- A table containing all matched relationships.

Access the table of nodes.

```
graphinfo.Nodes
```

```
ans = 

         NodeLabels      NodeData                NodeObject              
         __________    ____________    _______________________________

    0    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    1    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    2    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    3    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    4    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    5    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    6    'Person'      [1×1 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 =

         StartNodeID    RelationType    EndNodeID    RelationData
         ___________    ____________    _________    ____________

    1    0              'knows'         1            [1×1 struct]
    0    0              'knows'         2            [1×1 struct]
    3    1              'knows'         3            [1×1 struct]
    2    2              'knows'         1            [1×1 struct]
    5    3              'knows'         4            [1×1 struct]
    4    3              'knows'         5            [1×1 struct]
    6    5              'knows'         4            [1×1 struct]
    7    5              'knows'         6            [1×1 struct]
```

Access property keys for the first relationship.

```
graphinfo.Relations.RelationData{1}
```

```
ans = 

  struct with no fields.
```

The first relationship has no property keys.

Search the graph for all node labels in the database.

```
allnodes = nodeLabels(neo4jconn);
```

```
graphinfo = searchGraph(neo4jconn,allnodes);
```

Search Graph by 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 the graph for the relationship type `'knows'` using the Neo4j® database connection `neo4jconn`.

```
reltype = {'knows'};

graphinfo = searchGraph(neo4jconn,reltype)


graphinfo =

  struct with fields:

        Nodes: [7×3 table]
    Relations: [8×4 table]
```

`graphinfo` is a structure that contains the results of the search:

- `Nodes` -- A table containing all starting and ending nodes that denote each matched relationship.
- `Relations` -- A table containing all matched relationships.

Access the table of nodes.

```
graphinfo.Nodes
```

```
ans = 

         NodeLabels      NodeData                NodeObject            
         __________    ____________    ________________________________

    0    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    2    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    1    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    3    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    5    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    4    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
    6    'Person'      [1×1 struct]    [1x1 database.neo4j.Neo4jNode]
```

Access the table of relationships.

```
graphinfo.Relations
```

```
ans = 

         StartNodeID    RelationType    EndNodeID    RelationData
         ___________    ____________    _________    ____________

    0    0              'knows'         2            [1×1 struct]
    1    0              'knows'         1            [1×1 struct]
    2    2              'knows'         1            [1×1 struct]
    3    1              'knows'         3            [1×1 struct]
    4    3              'knows'         5            [1×1 struct]
    5    3              'knows'         4            [1×1 struct]
    6    5              'knows'         4            [1×1 struct]
    7    5              'knows'         6            [1×1 struct]
```

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 using the function neo4j.

criteria — Search criteria

cell array of character vectors

Search criteria, specified as a cell array of character vectors. 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.

Data Types: cell

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	Table that contains node information for each node in the Relations table. The Nodes table contains: • Row names in the table, which are Neo4j node identifiers of the matched database nodes. • The variable NodeLabels, which is a character vector that denotes the node label for each matched database node.

Field	Description
	• The variable `NodeData`, which is a structure array that contains node information such as property keys for each matched database node. • The variable `NodeObject`, which is the `Neo4jNode` object for each matched database node. If `criteria` contains node labels, the output is automatically sorted by `StartNodeID` and `Label`.
`Relations`	Table that contains relationship information for the nodes in the `Nodes` table. The `Relations` table contains: • Row names in the table, which are Neo4j relationship identifiers. • The variable `StartNodeID`, which is the node identifier for the start node for each matched relationship. • The variable `RelationType`, which is a character vector that denotes the relationship type for each matched relationship. • The variable `EndNodeID`, which is the node identifier for the end node for each matched relationship. • The variable `RelationData`, which is a structure array that contains property keys associated with each matched relationship. If `criteria` contains relationship types, the output is automatically sorted by `RelationID`.

More About

- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 7-10
- “Working with the MATLAB Interface to Neo4j” on page 7-8

See Also

`neo4j` | `nodeLabels` | `relationTypes` | `searchNode` | `searchNodeByID` | `searchRelation`

Introduced in R2016b

executeCypher

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 using the function `neo4j`.

`query` — Cypher query

character vector

Cypher query, specified as a character vector.

Example: `'MATCH (movie: Movie {title: ''The Matrix''}) RETURN movie.title, movie.studio'`

Data Types: `char`

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.

More About

- “Working with the MATLAB Interface to Neo4j” on page 7-8
- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 7-10
- “MATLAB Interface to Neo4j Error Messages” on page 7-13

See Also

`neo4j` | `searchGraph` | `searchNode` | `searchNodeByID` | `searchRelation`

Introduced in R2016b

nodeRelationTypes

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 7-2

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 one of these values. The relationships are associated with the specified origin node.

Value	Description
`'in'`	Incoming relationship
`'out'`	Outgoing relationship

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.

More About

- "Working with the MATLAB Interface to Neo4j" on page 7-8
- "Searching Graph Database Using MATLAB Interface to Neo4j" on page 7-10

See Also

`nodeDegree` | `searchNode` | `searchNodeByID`

Introduced in R2016b

nodeDegree

In- 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 7-2

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 one of these values. The relationships are associated with the specified origin node.

Value	Description
`'in'`	Incoming relationship
`'out'`	Outgoing relationship

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.

More About

- “Working with the MATLAB Interface to Neo4j” on page 7-8
- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 7-10

See Also

`nodeRelationTypes` | `searchNode` | `searchNodeByID`

Introduced in R2016b

neo4jStruct2Digraph

Convert graph or relationship structure from Neo4j database to directed graph

Syntax

```
G = neo4jStruct2Digraph(s)
G = neo4jStruct2Digraph(s,Name,Value)
```

Description

`G = neo4jStruct2Digraph(s)` creates a directed graph from the structure `s`. With the directed graph, run graph network analytics using MATLAB. For example, to visualize the graph, see "Graph Plotting and Customization".

`G = neo4jStruct2Digraph(s,Name,Value)` creates a directed graph with additional options specified by the `Name,Value` pair arguments.

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'      [1×1 struct]
    '1'     'Person'      [1×1 struct]
    '2'     'Person'      [1×1 struct]
    '3'     'Person'      [1×1 struct]
    '4'     'Person'      [1×1 struct]
    '5'     'Person'      [1×1 struct]
    '6'     'Person'      [1×1 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 `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 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'      [1×1 struct]
    'User3'    'Person'      [1×1 struct]
    'User2'    'Person'      [1×1 struct]
    'User4'    'Person'      [1×1 struct]
    'User5'    'Person'      [1×1 struct]
    'User6'    'Person'      [1×1 struct]
    'User7'    'Person'      [1×1 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`

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: `G = neo4jStruct2Digraph(graphinfo,'NodeNames',nodenames);`

`'NodeNames'` — Neo4j database node names

cell array of character vectors

Neo4j database node names, specified as the comma-separated pair consisting of `'NodeNames'` and a cell array of character vectors. To add the Neo4j database node names to the directed graph, specify this name-value pair argument.

Example: `'NodeNames',nodenames`

Data Types: `cell`

Output Arguments

`G` — Directed graph

`digraph` object

Directed graph, returned as a `digraph` object.

More About

- “Searching Graph Database Using MATLAB Interface to Neo4j” on page 7-10
- “Working with the MATLAB Interface to Neo4j” on page 7-8

See Also

`distances` | `neo4j` | `searchGraph` | `searchNode` | `searchRelation`

Introduced in R2016b