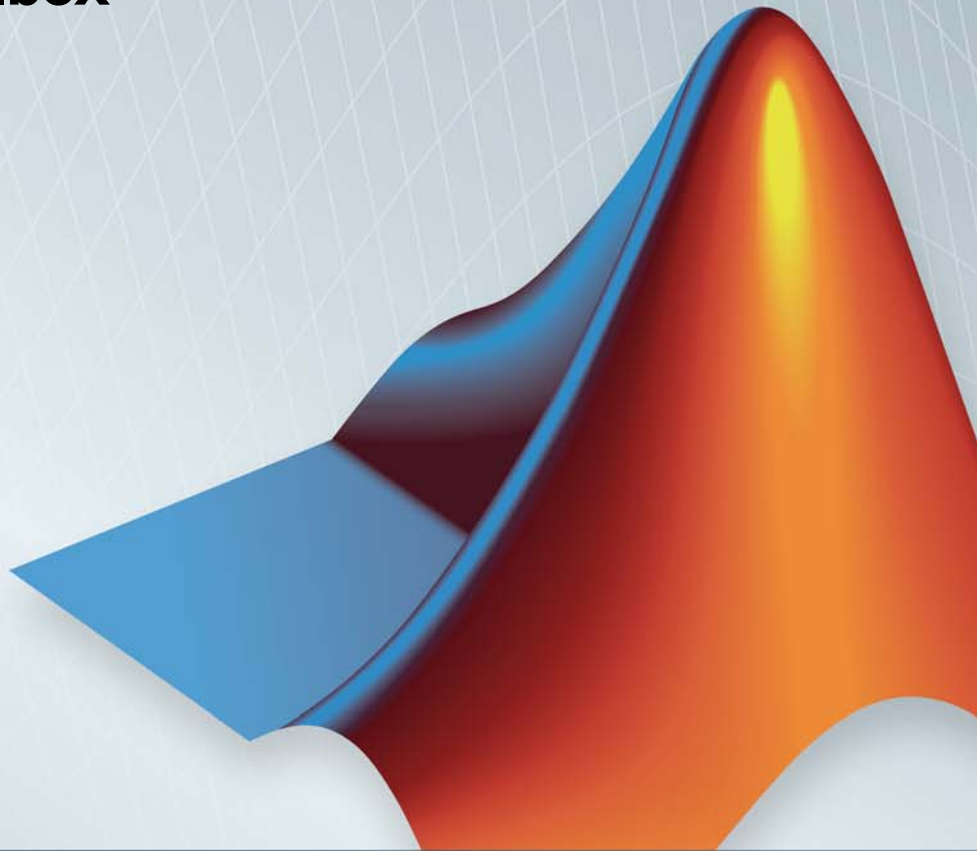


# Database Toolbox™

## User's Guide

R2013b



# MATLAB®



## How to Contact MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto:info@mathworks.com) Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

For contact information about worldwide offices, see the MathWorks Web site.

*Database Toolbox™ User's Guide*

© COPYRIGHT 1998–2013 by The MathWorks, Inc.

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

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

### Trademarks

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

### Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

May 1998	Online Only	New for Version 1 for MATLAB® 5.2
July 1998	First Printing	For Version 1
Online only	June 1999	Revised for Version 2 (Release 11)
December 1999	Second printing	For Version 2 (Release 11)
Online only	September 2000	Revised for Version 2.1 (Release 12)
June 2001	Third printing	Revised for Version 2.2 (Release 12.1)
July 2002	Online only	Revised for Version 2.2.1 (Release 13)
November 2002	Fourth printing	Version 2.2.1
June 2004	Fifth printing	Revised for Version 3.0 (Release 14)
October 2004	Online only	Revised for Version 3.1 (Release 14SP1)
March 2005	Online only	Revised for Version 3.0.2 (Release 14SP2)
September 2005	Online only	Revised for Version 3.1 (Release 14SP3)
March 2006	Online only	Revised for Version 3.1.1 (Release 2006a)
September 2006	Online only	Revised for Version 3.2 (Release 2006b)
October 2006	Sixth printing	Revised for Version 3.2 (Release 2006b)
March 2007	Online only	Revised for Version 3.3 (Release 2007a)
September 2007	Seventh printing	Revised for Version 3.4 (Release 2007b)
March 2008	Online only	Revised for Version 3.4.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.5 (Release 2008b)
March 2009	Online only	Revised for Version 3.5.1 (Release 2009a)
September 2009	Online only	Revised for Version 3.6 (Release 2009b)
March 2010	Online only	Revised for Version 3.7 (Release 2010a)
September 2010	Online only	Revised for Version 3.8 (Release 2010b)
reApril 2011	Online only	Revised for Version 3.9 (Release 2011a)
September 2011	Online only	Revised for Version 3.10 (Release 2011b)
March 2012	Online only	Revised for Version 3.11 (Release 2012a)
September 2012	Online only	Revised for Version 4.0 (Release 2012b)
March 2013	Online only	Revised for Version 4.1 (Release 2013a)
September 2013	Online only	Revised for Version 5.0 (Release 2013b)



## Before You Begin

### 1

<b>Working with Databases</b> .....	1-2
Connecting to Databases .....	1-2
Platform Support .....	1-2
Database Support .....	1-2
Driver Support .....	1-3
Structured Query Language (SQL) .....	1-4
<b>Data Type Support</b> .....	1-5
<b>Data Retrieval Restrictions</b> .....	1-7
Spaces in Table Names or Column Names .....	1-7
Quotation Marks in Table Names or Column Names .....	1-7
Reserved Words in Column Names .....	1-7

## Working with Data Sources

### 2

<b>Setting Up ODBC Data Sources</b> .....	2-2
<b>Setting Up JDBC Data Sources</b> .....	2-3
<b>Accessing Existing JDBC Data Sources</b> .....	2-4
<b>Modifying Existing JDBC Data Sources</b> .....	2-5
<b>Removing JDBC Data Sources</b> .....	2-6
<b>Troubleshooting JDBC Driver Problems</b> .....	2-7

<b>Database Connection Error Messages</b> .....	<b>2-8</b>
<b>Database Explorer Error Messages</b> .....	<b>2-10</b>
<b>Using the Native ODBC Database Connection</b> .....	<b>2-12</b>
About the Native ODBC Interface .....	<b>2-12</b>
Native ODBC Interface Workflow .....	<b>2-12</b>
Native ODBC, ODBC/JDBC Bridge and JDBC Interface Comparison .....	<b>2-15</b>
Compatibility and Limitations .....	<b>2-17</b>

## Using Visual Query Builder

# 3

<b>Getting Started with Visual Query Builder</b> .....	<b>3-2</b>
What Is Visual Query Builder? .....	<b>3-2</b>
Using Queries to Import Data .....	<b>3-2</b>
Using Queries to Export Data .....	<b>3-4</b>
<b>Working with Preferences</b> .....	<b>3-6</b>
Specifying Preferences .....	<b>3-6</b>
<b>Preference Settings for Large Data Import</b> .....	<b>3-10</b>
Will All Data (Size $n$ ) Fit in a MATLAB Variable? .....	<b>3-11</b>
Will All of This Data Fit in the JVM Heap? .....	<b>3-12</b>
How Do I Perform Batching? .....	<b>3-12</b>
<b>Displaying Query Results</b> .....	<b>3-15</b>
How to Display Query Results .....	<b>3-15</b>
Displaying Data Relationally .....	<b>3-15</b>
Charting Query Results .....	<b>3-19</b>
Displaying Query Results in an HTML Report .....	<b>3-21</b>
Displaying Query Results with MATLAB Report Generator .....	<b>3-22</b>
<b>Fine-Tuning Queries Using Advanced Query Options</b> .....	<b>3-27</b>

Retrieving All Occurrences vs. Unique Occurrences of Data .....	3-27
Retrieving Data That Meets Specified Criteria .....	3-29
Grouping Statements .....	3-32
Displaying Results in a Specified Order .....	3-36
Using Having Clauses to Refine Group by Results .....	3-39
Creating Subqueries for Values from Multiple Tables .....	3-42
Creating Queries That Include Results from Multiple Tables .....	3-47
Additional Advanced Query Options .....	3-50
<b>Retrieving BINARY and OTHER Data Types .....</b>	<b>3-51</b>
<b>Importing and Exporting BOOLEAN Data .....</b>	<b>3-54</b>
Importing BOOLEAN Data from Databases .....	3-54
Exporting BOOLEAN Data to Databases .....	3-57
<b>Saving Queries in Files .....</b>	<b>3-59</b>
About Generated Files .....	3-59
VQB Query Elements in Generated Files .....	3-60
<b>Using Database Explorer .....</b>	<b>3-61</b>
About Database Explorer .....	3-61
Workflow .....	3-62
Configure Your Environment .....	3-62
Database Connection Error Messages .....	3-74
Set Database Preferences .....	3-76
Display Data from a Single Database Table .....	3-78
Join Data from Multiple Database Tables .....	3-80
Define Query Criteria to Refine Results .....	3-85
Query Rules Using the SQL Criteria Panel .....	3-87
Query Example Using a Left Outer Join .....	3-89
Work with Multiple Databases .....	3-98
Import Data to the MATLAB Workspace .....	3-98
Save Queries as SQL Code .....	3-101
Generate MATLAB Code .....	3-102

## Using Database Toolbox Functions

# 4

<b>Getting Started with Database Toolbox Functions</b> . . . . .	<b>4-2</b>
<b>Importing Data from Databases</b> . . . . .	<b>4-3</b>
<b>Viewing Information About Imported Data</b> . . . . .	<b>4-5</b>
<b>Exporting Data to New Record in Database</b> . . . . .	<b>4-8</b>
<b>Replacing Existing Database Data with Exported Data</b> . . . . .	<b>4-12</b>
<b>Exporting Multiple Records from the MATLAB Workspace</b> . . . . .	<b>4-14</b>
<b>Exporting Data Using the Bulk Insert Command</b> . . . . .	<b>4-18</b>
Bulk Insert to Oracle . . . . .	<b>4-18</b>
Bulk Insert to Microsoft SQL Server 2005 . . . . .	<b>4-20</b>
Bulk Insert to MySQL . . . . .	<b>4-22</b>
<b>Retrieving Image Data Types</b> . . . . .	<b>4-25</b>
<b>Working with Database Metadata</b> . . . . .	<b>4-27</b>
Accessing Metadata . . . . .	<b>4-27</b>
Resultset Metadata Objects . . . . .	<b>4-33</b>
<b>Using Driver Functions</b> . . . . .	<b>4-34</b>
<b>About Database Toolbox Objects and Methods</b> . . . . .	<b>4-36</b>
<b>Using the exec Function</b> . . . . .	<b>4-39</b>
About the exec Function . . . . .	<b>4-39</b>
Using Cursor Objects . . . . .	<b>4-39</b>
Working with Microsoft Excel . . . . .	<b>4-40</b>
Database Considerations . . . . .	<b>4-40</b>



**Using the fetch Function** ..... 4-42  
  About the fetch Function ..... 4-42  
  fetch Workflow ..... 4-42  
  Using fetch with a Cursor Object ..... 4-43  
  Database Considerations ..... 4-44

**Functions — Alphabetical List**

**5**

**Index**



# Before You Begin

---

- “Working with Databases” on page 1-2
- “Data Type Support” on page 1-5
- “Data Retrieval Restrictions” on page 1-7

## Working with Databases

In this section...
“Connecting to Databases” on page 1-2
“Platform Support” on page 1-2
“Database Support” on page 1-2
“Driver Support” on page 1-3
“Structured Query Language (SQL)” on page 1-4

### Connecting to Databases

Before you can use this toolbox to connect to a database, you must set up data sources. For more information, see “Configuring Your Environment”.

### Platform Support

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

For more information, 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-112010-0>

## **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 more information about how Database Toolbox handles BOOLEAN data, see “Importing and Exporting BOOLEAN Data” on page 3-54.

---

- 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.

---



## Data Retrieval Restrictions

In this section...
“Spaces in Table Names or Column Names” on page 1-7
“Quotation Marks in Table Names or Column Names” on page 1-7
“Reserved Words in Column Names” on page 1-7

### 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.



# Working with Data Sources

---

- “Setting Up ODBC Data Sources” on page 2-2
- “Setting Up JDBC Data Sources” on page 2-3
- “Accessing Existing JDBC Data Sources” on page 2-4
- “Modifying Existing JDBC Data Sources” on page 2-5
- “Removing JDBC Data Sources” on page 2-6
- “Troubleshooting JDBC Driver Problems” on page 2-7
- “Database Connection Error Messages” on page 2-8
- “Database Explorer Error Messages” on page 2-10
- “Using the Native ODBC Database Connection” on page 2-12

## **Setting Up ODBC Data Sources**

For instructions on setting up ODBC data sources, see “Configure ODBC Data Sources”.

## Setting Up JDBC Data Sources

For instructions on setting up JDBC data sources, see “Configure JDBC Data Sources”.

## 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 2-4.
- 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 2-4.
- 2** Click **Remove**.
- 3** Click **OK** to save your changes and close the Define JDBC data sources dialog box.



## Troubleshooting JDBC Driver Problems

This section describes how to address common data source access problems, in which selecting a data source in the Visual Query Builder list produces an error, or the data source is not in the list as expected. There are several potential causes for these issues:

- The database is unavailable, or there are connectivity problems. Try selecting the data source in VQB again. If you are still unable to access the data source, contact your database administrator.
- You ran the `clear all` command in the MATLAB Command Window after you defined a JDBC data source. In this case, redefine the data source by following the instructions in “Configure JDBC Data Sources”.

## Database Connection Error Messages

### Connection Error Messages and Probable Causes

Vendor	Error Message	Probable Causes
All	Unable to find JDBC driver.	<ul style="list-style-type: none"> <li>• Path to the JDBC driver jar file is not on the static or dynamic class path.</li> <li>• Incorrect driver name provided while using the 'driver' and 'url' syntax.</li> </ul>
All	[Microsoft][ODBC Driver Manager] The specified DSN contains an architecture mismatch between Driver and Application	You tried to open a 32-bit application when running MATLAB in 64-bit mode. Restart MATLAB to run in 32-bit mode using the command <code>matlab win32</code> .
Microsoft SQL Server	The TCP/IP connection to the host <code>hostname</code> , port <code>portnumber</code> 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. Microsoft SQL Server uses a dynamic port for JDBC and the value should be verified using Microsoft SQL Server Configuration Manager.
Microsoft SQL Server	This driver is not configured for integrated authentication.	The Microsoft SQL Server Windows authentication library is not added to <code>librarypath.txt</code> . For more information, see the database example for Microsoft SQL Server Authenticated Database Connection.

**Connection Error Messages and Probable Causes (Continued)**

Vendor	Error Message	Probable Causes
Microsoft SQL Server	Invalid string or buffer length.	64-bit ODBC driver error. Use a JDBC driver or the native ODBC interface instead.
MySQL	Access denied for user 'user'@'machinename' (using password: YES)	Incorrect user name and password combination.
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.
MySQL	Unknown database 'databasename'.	Provided database name is incorrect.
Oracle	Error when connecting to Oracle oci8 database using JDBC driver:  Error using com.mathworks.toolbox.database.database. Java exception occurred: java.lang.UnsatisfiedLinkError: no ocijdbc11 in java.library.pathat java.lang.ClassLoader.loadLibrary(Unknown Source)at java.lang.Runtime.loadLibrary0.....	MATLAB cannot find the Oracle DLL that the oci8 drivers need. To correct the problem, add the path for the location of the Oracle DLLs to \$MATLAB/toolbox/local/librarypath.txt.
Oracle	Invalid Oracle URL specified:  OracleDataSource.makeURL	DriverType parameter is not specified.
Oracle	The Network Adapter could not establish the connection.	Either Server or Portnumber is not specified or has an incorrect value.

**See Also** database

## Database Explorer Error Messages

### Database Explorer Error Messages and Probable Causes

Vendor	Position of Error	Error Message	Probable Causes
All	Error occurs in the <b>Connection Failure</b> dialog box after clicking <b>Connect</b> in the <b>Connect to a data source</b> 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 the following command: <code>setdbprefs(... 'JDBCDataSourceFile', '')</code> and then create a new JDBC data source from Database Explorer.
All	Error occurs in the <b>Connect to a data source</b> dialog box.	[Microsoft][ODBC Driver Manager] The specified DSN contains an architecture mismatch between Driver and Application	You tried to open a 32-bit application when running MATLAB in 64-bit mode. Restart MATLAB to run in 32-bit mode using the command <code>matlab win32</code> .
Microsoft Access	Error occurs in the <b>Connection Failure</b> dialog box after clicking <b>Connect</b> in the <b>Connect to a data source</b> dialog box.	[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	The file location of the Microsoft Access database is incorrect. Verify the location of the database file and modify the existing file location by selecting <b>New &gt; ODBC</b> and selecting the existing database name from <b>ODBC Data Source Administrator</b> dialog box. Then select <b>Configure</b> to change the database file location.

**Database Explorer Error Messages and Probable Causes (Continued)**

<b>Vendor</b>	<b>Position of Error</b>	<b>Error Message</b>	<b>Probable Causes</b>
Microsoft SQL Server	Error occurs in the <b>Data Preview Error</b> dialog box after selecting a column of a table in the Database Browser pane.	Invalid Object Name <i>catalog name.table name</i>	You must select the appropriate schema name in Database Explorer using the <b>Catalog/Schema</b> address bar above the table columns tree.
Oracle	Error occurs inside the Database Browser pane.	No tables found in this schema Consider changing the schema.	This error occurs when using the Oracle ODBC driver because of a problem in the JDBC-ODBC bridge. Please switch your database connection to use a JDBC driver. For more information, see “Configure JDBC Data Sources”.

## Using the Native ODBC Database Connection

In this section...
“About the Native ODBC Interface” on page 2-12
“Native ODBC Interface Workflow” on page 2-12
“Native ODBC, ODBC/JDBC Bridge and JDBC Interface Comparison” on page 2-15
“Compatibility and Limitations” on page 2-17

### 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 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, execute an SQL statement and fetch the returned data, insert data, and then 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, you must 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`.

```
insert(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.

---

Close the cursor object, `curs`, and then close the database connection, `conn`.

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



**Caution:** Leaving cursor and connection objects open or overwriting open objects can result in unexpected behavior. Once you are finished working with these objects, you must close them using `close`.

## Native ODBC, ODBC/JDBC Bridge and JDBC Interface Comparison

This table highlights the differences between using the native ODBC, ODBC/JDBC bridge, and JDBC interfaces to access and manipulate data in a database.

Item	Native ODBC	ODBC/JDBC Bridge	JDBC
Connection function	Use <code>database.ODBCConnection</code>	Use database	Use database
Actions	<p>Can perform the following actions:</p> <ul style="list-style-type: none"> <li>• Query data (<code>exec</code>)</li> <li>• Import data (<code>fetch</code>)</li> <li>• Run stored procedure (<code>exec</code>)</li> <li>• Export data (<code>insert, fastinsert</code>)</li> <li>• Close connection (<code>close</code>)</li> </ul>	<p>Can perform the following actions:</p> <ul style="list-style-type: none"> <li>• Query data (<code>exec</code>)</li> <li>• Import data (<code>fetch</code>)</li> <li>• Export data (<code>insert, fastinsert, datainsert, update</code>)</li> <li>• Run stored procedure (<code>exec, runstoredprocedure</code>)</li> <li>• Retrieve metadata (<code>dmd, tables, columns,</code></li> </ul>	<p>Can perform the following actions:</p> <ul style="list-style-type: none"> <li>• Query data (<code>exec</code>)</li> <li>• Import data (<code>fetch</code>)</li> <li>• Export data (<code>insert, fastinsert, datainsert, update</code>)</li> <li>• Run stored procedure (<code>exec, runstoredprocedure</code>)</li> <li>• Retrieve metadata (<code>dmd, tables, columns,</code></li> </ul>

Item	Native ODBC	ODBC/JDBC Bridge	JDBC
		database.catalogs, and many others) <ul style="list-style-type: none"> <li>• Use Database Explorer (dexplore)</li> <li>• Close connection (close)</li> </ul>	database.catalogs, and many others) <ul style="list-style-type: none"> <li>• Use Database Explorer (dexplore)</li> <li>• Close connection (close)</li> </ul>
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	Slowest	Medium
64-bit systems	No major issues	Several known issues with connectivity and data access	No major issues
Data type support	Long data types are not supported (e.g. LONG, BLOB, etc.)	Long data types are supported	Long data types are supported

---

**Note** For more information about the `database.ODBCConnection` syntax, see the native ODBC interface example in `database`.

---

## Compatibility and Limitations

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

- Native ODBC database connections are supported on MATLAB 32-bit and 64-bit versions using the `database` function. The native ODBC interface supports 64-bit database vendors. This interface is backward compatible for 32-bit versions. The bitness of MATLAB must always match the bitness of the database driver.
- 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 long data types such as Oracle LONG and SQL Server NTEXT. If you get one of the following errors, 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.

## Concepts

- “Using the exec Function” on page 4-39
- “Using the fetch Function” on page 4-42



# Using Visual Query Builder

---

- “Getting Started with Visual Query Builder” on page 3-2
- “Working with Preferences” on page 3-6
- “Preference Settings for Large Data Import” on page 3-10
- “Displaying Query Results” on page 3-15
- “Fine-Tuning Queries Using Advanced Query Options” on page 3-27
- “Retrieving BINARY and OTHER Data Types” on page 3-51
- “Importing and Exporting BOOLEAN Data” on page 3-54
- “Saving Queries in Files” on page 3-59
- “Using Database Explorer” on page 3-61

## Getting Started with Visual Query Builder

In this section...
“What Is Visual Query Builder?” on page 3-2
“Using Queries to Import Data” on page 3-2
“Using Queries to Export Data” on page 3-4

### 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

1\* Specify **Select**.      2\* Select data source.      3 Select catalog and schema.      4\* Select tables.      5\* Select fields to retrieve.

12 View query results in table, chart, and report formats.

8 Set preferences for data retrieval.

13 Save, load, and run queries, and generate M-files.

6 Refine query.

7 View SQL statement.

9\* Assign variable for results.

11 Double-click to view query results in MATLAB Array Editor.      10\* Run query.

**Visual Query Builder**

Query Display Help

Data operation  
 Select     Insert

Data source      Catalog      Tables      Fields

MIS Access Databases  
 SampleDB  
 dBASE Files  
 dbtoolboxdemo

<default>  
 Schema  
 <default>

inventoryTable  
 productTable  
 salesVolume  
 suppliers  
 Temperatures

StockNumber  
 January  
 February  
 March  
 April

Advanced query options  
 All    Where...    Group by...    Having...    Order by...  
 Distinct    > 400000

SQL statement  
 SELECT ALL StockNumber, March FROM salesVolume WHERE StockNumber > 400000

MATLAB workspace variable  
 A    Execute

Data  
 Workspace variable    Size    Memory (bytes)

Workspace variable	Size	Memory (bytes)
A	7x2	952

For a step-by-step example of how to use queries to import data into the MATLAB workspace from a database, see “Using Queries to Import Database Data”.

### **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

The screenshot shows the Visual Query Builder window with the following configuration:

- Data operation:**  Insert
- Data source:** dBASE Files
- Catalog:** <default>
- Schema:** <default>
- Tables:** Avg\_Freight\_Cost, Categories, Customers, Employees
- Fields:** Calc\_Date, Avg\_Cost
- Advanced query options:**  All,  Distinct
- MATLAB command:** `insert(conn,'Avg_Freight_Cost',{'Calc_Date','Avg_Cost'},export_data)`
- MATLAB workspace variable:** export\_data
- Data table:**

Workspace variable	Size	Memory (bytes)
export_data	1x2	150

Annotations in the image:

- 1\* Specify **Insert**.
- 2\* Select data source.
- 3 Select catalog and schema.
- 4\* Select tables.
- 5\* Select fields to which to export data.
- 6\* Specify variable containing data to export.
- 7 View MATLAB statement.
- 8\* Run query.
- 9 Save, load, and run queries, set preferences for exporting NULLs, and generate M-files.

For a step-by-step example of how to use queries to export data from the MATLAB workspace to a database, see “Using Queries to Export Data to Databases”.

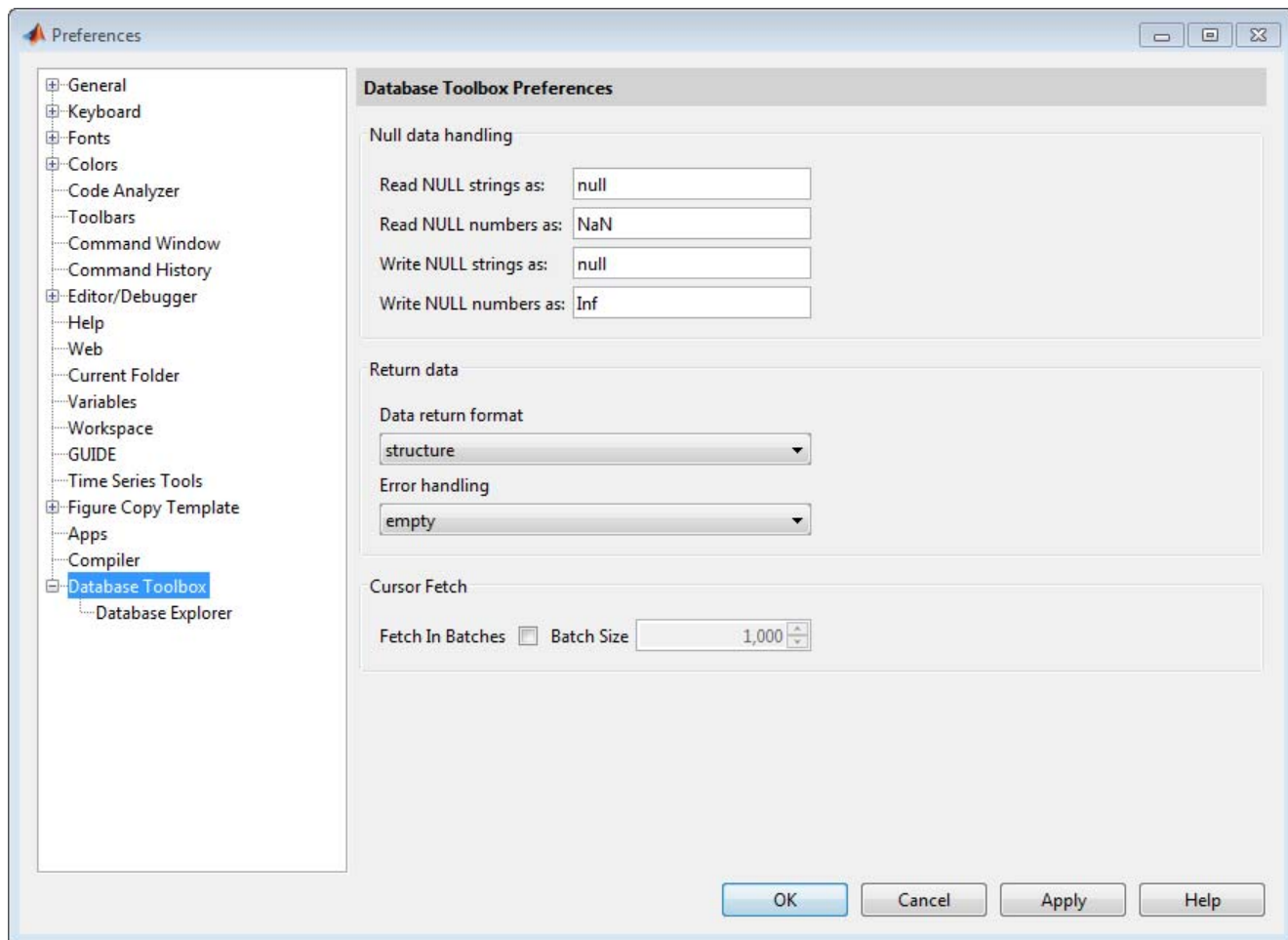
## Working with Preferences

### Specifying 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 the following table.

<b>Preference</b>	<b>Acceptable Values</b>	<b>Description</b>
<b>Read NULL strings as:</b>	null (default)	Specifies how NULL strings appear after being fetched from a database.
<b>Read NULL numbers as:</b>	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 as NaN. Setting this field to 0 causes NULL data imported into the MATLAB workspace to appear as 0s.
<b>Write NULL strings as:</b>	null (default)	Specifies how NULL strings appear after being exported to a database. This setting does not apply to Database Explorer (dexplore).
<b>Write NULL numbers as:</b>	Nan (default)	Specifies how NULL numbers appear after being exported to a database. This setting does not apply to Database Explorer (dexplore).
<b>Data return format</b>	cell array, numeric, structure, or dataset	<p>Select a data format based on the type of data you are importing, memory considerations, and your preferred method of working with retrieved data.</p> <ul style="list-style-type: none"> <li>• <b>cellarray</b> (default) — Imports nonnumeric data into MATLAB cell arrays.</li> <li>• <b>numeric</b> — Imports data into MATLAB matrix of doubles. Nonnumeric data types are considered NULL and appear as specified in the <b>Read NULL numbers as:</b> setting. Use only when data to retrieve is in numeric format, or when nonnumeric data to retrieve is not relevant.</li> <li>• <b>structure</b> — Imports data into a MATLAB structure. Use for all data types. Facilitates working with returned columns.</li> <li>• <b>dataset</b> — Imports data into MATLAB dataset objects. This option requires Statistics Toolbox™.</li> </ul> <p>This setting does not apply to Database Explorer (dexplore). If you are using Database Explorer, the data return format is</p>

Preference	Acceptable Values	Description
		specified using <b>Imported Data</b> panel in the Database Explorer interface.
<b>Error handling</b>	store, report, or empty	<ul style="list-style-type: none"> <li>• Set this field to <b>store</b> or <b>empty</b> 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.</li> <li>• Set this field to <b>report</b> to display query errors in the MATLAB Command Window.</li> </ul> <p>This setting does not apply to Database Explorer (dexplore).</p>
<b>Cursor Fetch</b>	Fetch In Batches and Batch Size	<p>Specifies if <code>fetch</code> retrieves data in batches with a user-defined <b>Batch Size</b>. The default <b>Batch Size</b> is 1,000. For more information, see “Preference Settings for Large Data Import” on page 3-10.</p> <p>This setting does not apply to Database Explorer (dexplore). If you are using Database Explorer, the import batch size is specified using <b>Preferences</b> on the Database Explorer Toolstrip.</p>

- 3** Click **OK**. For more information about Preferences, see the `setdbprefs` function reference page.

## Preference Settings for Large Data Import

### In this section...

“Will All Data (Size  $n$ ) Fit in a MATLAB Variable?” on page 3-11

“Will All of This Data Fit in the JVM Heap?” on page 3-12

“How Do I Perform Batching?” on page 3-12

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 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:

```
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:

```
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:

```
results = runsqlscript(conn,'<filename>.sql')
```

## Displaying Query Results

### In this section...

“How to Display Query Results” on page 3-15

“Displaying Data Relationally” on page 3-15

“Charting Query Results” on page 3-19

“Displaying Query Results in an HTML Report” on page 3-21

“Displaying Query Results with MATLAB® Report Generator™” on page 3-22

### 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 MATLAB 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**.

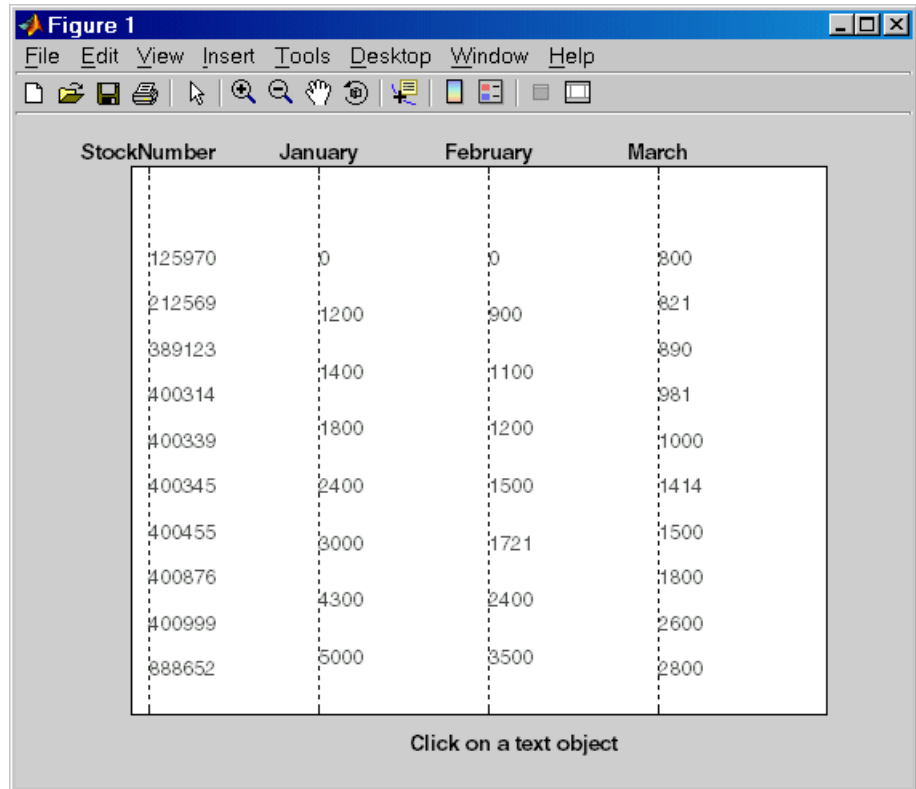
### Displaying Data Relationally

To display the results of `basic.qry`:

- 1** Execute `basic.qry`.

## 2 Select **Display > Data**.

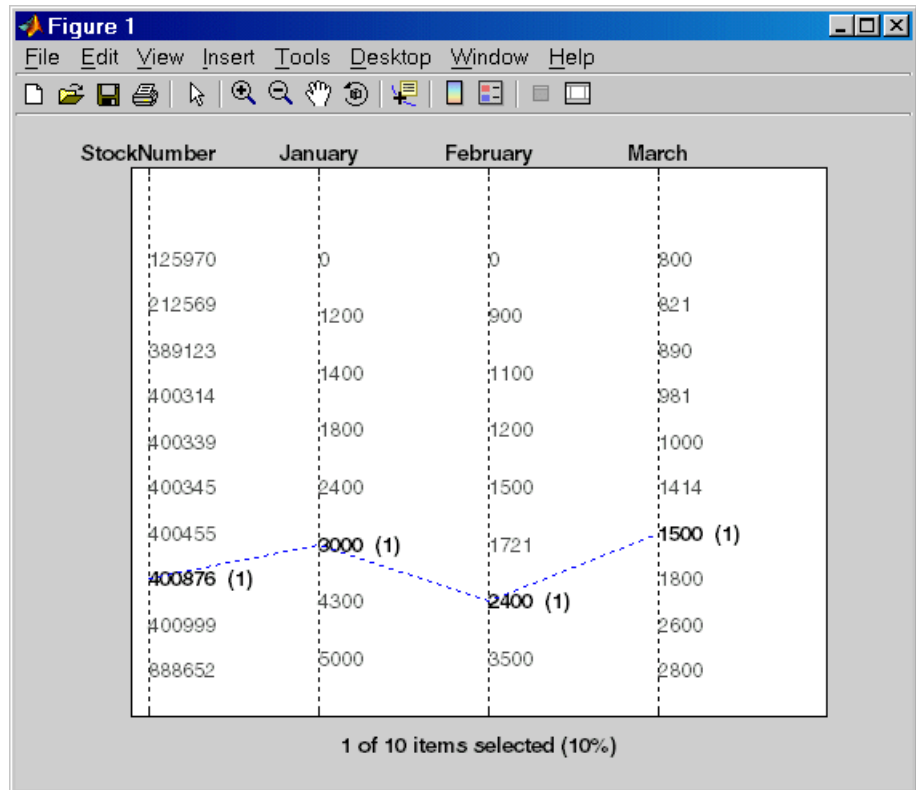
The query results appear in a figure window.



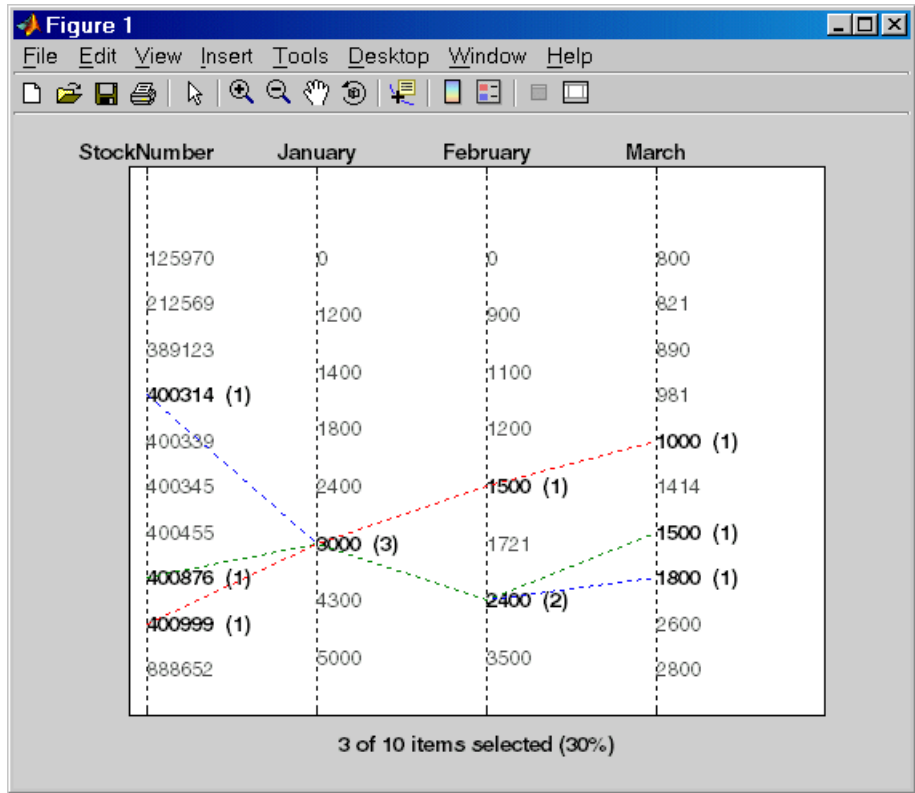
This display shows only unique values for each field, so you should not read each row as a single record. In this example, there are 10 entries for **StockNumber**, eight entries for **January** and **February**, and 10 entries for **March**. The number of entries in each field corresponds to the number 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.

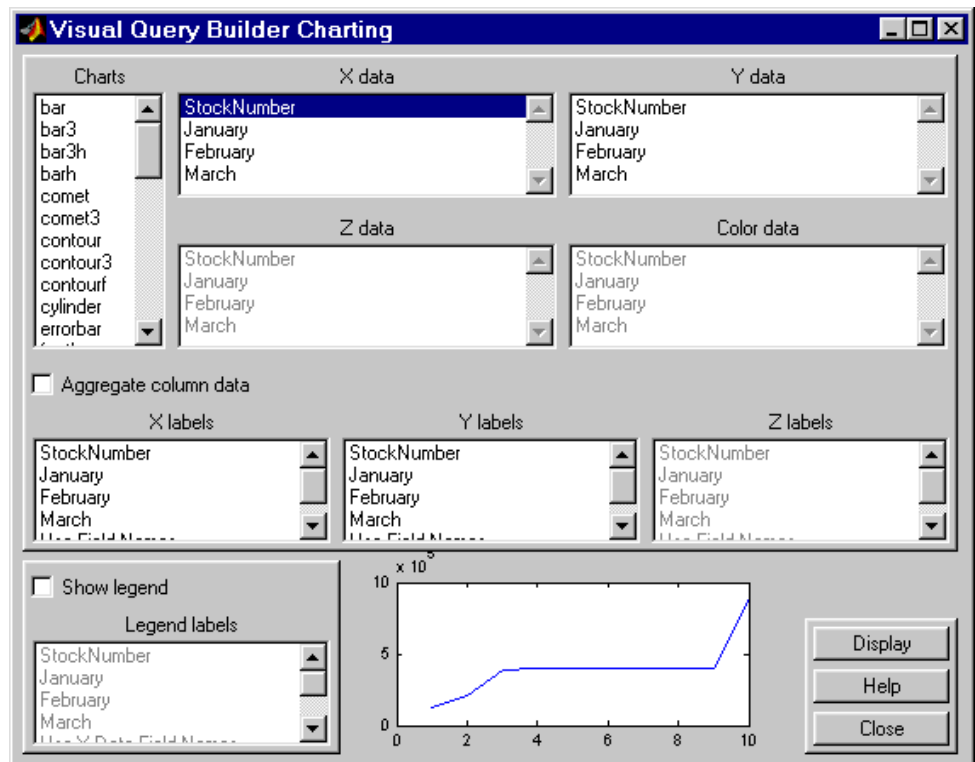


## Charting Query Results

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.

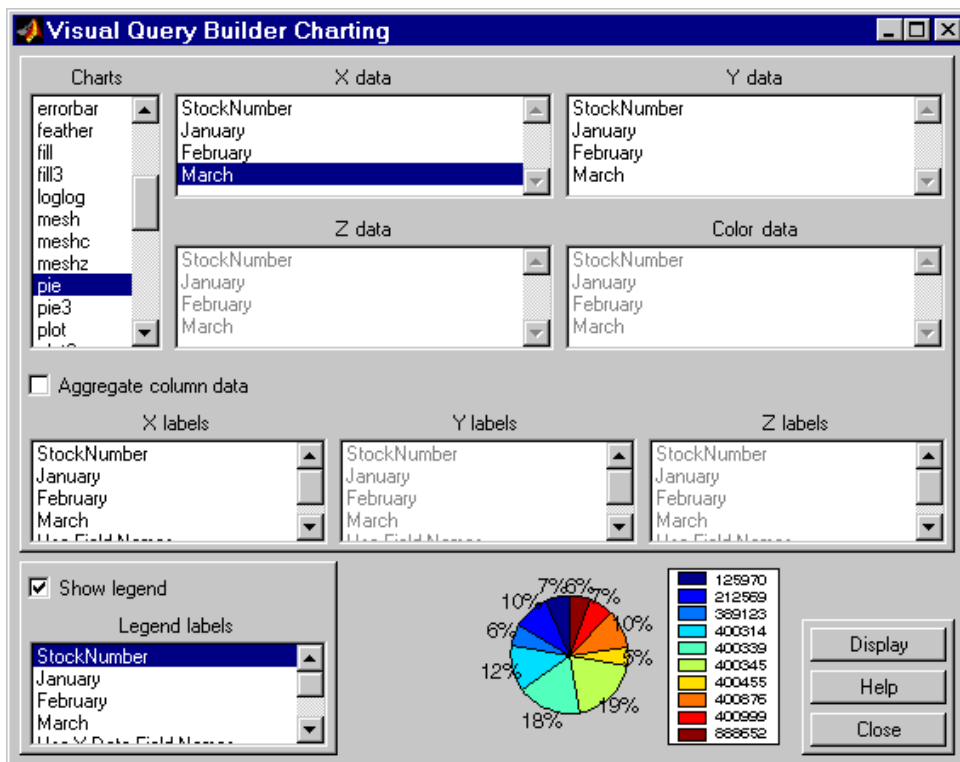
The pie chart preview now shows percentages for March data.

- 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.

Web Browser - file:///C:/Users/tflessa/databasetlbx.html

databasetlbx.html x +

Location: file:///C:/Users/tflessa/databasetlbx.html

**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](#)

---

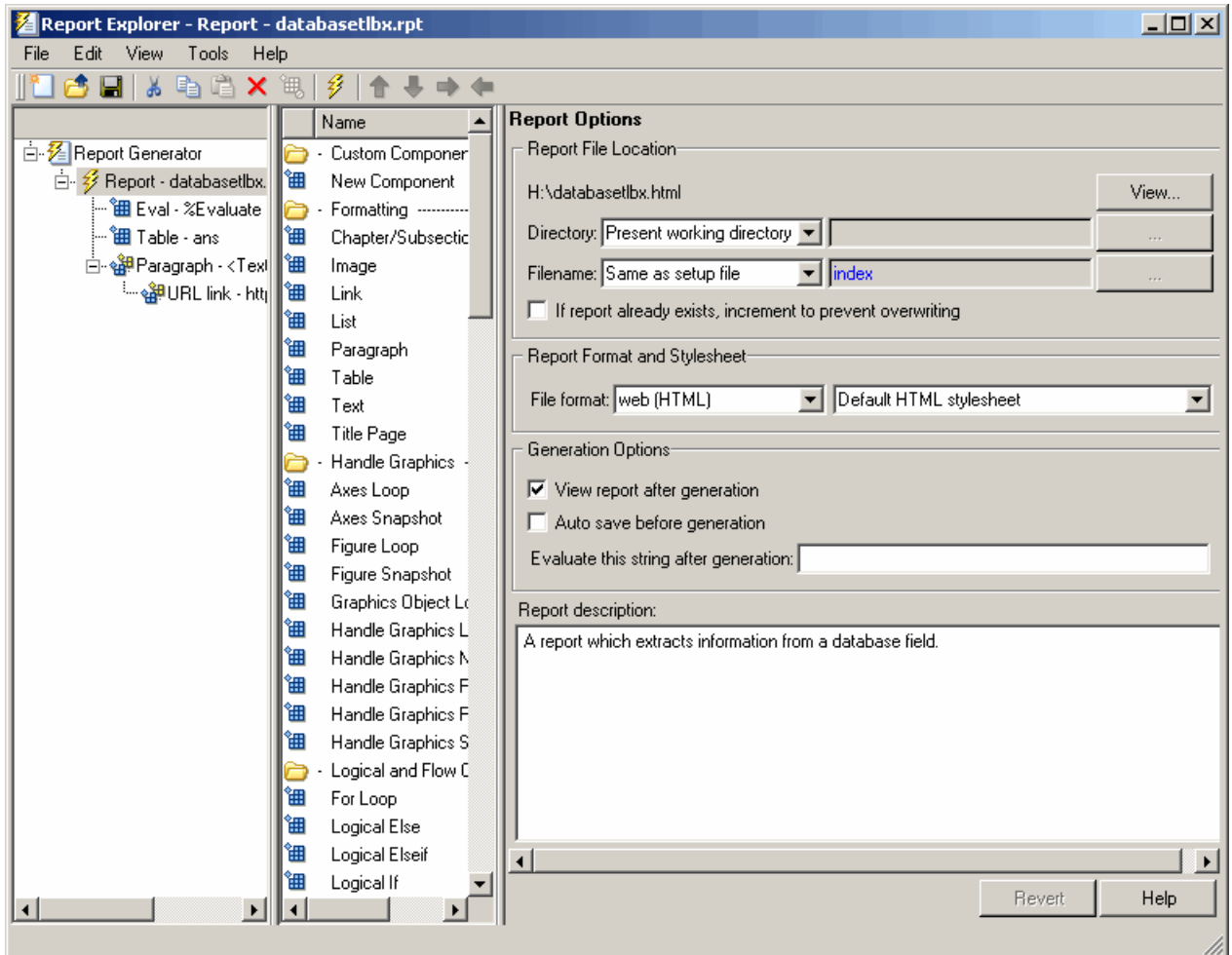
**Tip** Because some browsers do not start automatically, you may need to open your Web browser before displaying the query results.

---

## **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/databaset1bx.rpt` from the Options pane in the middle of the Report Explorer window.

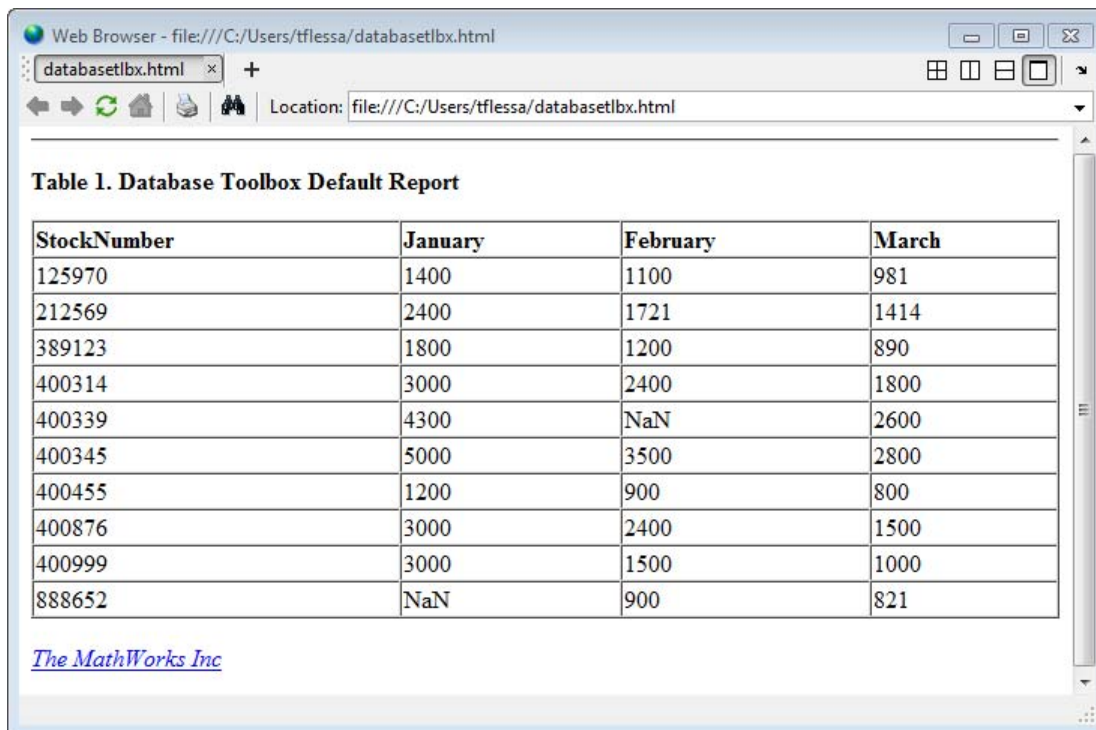


**3** Open the report template for editing by clicking **Open a Report file or stylesheet.**

- 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.

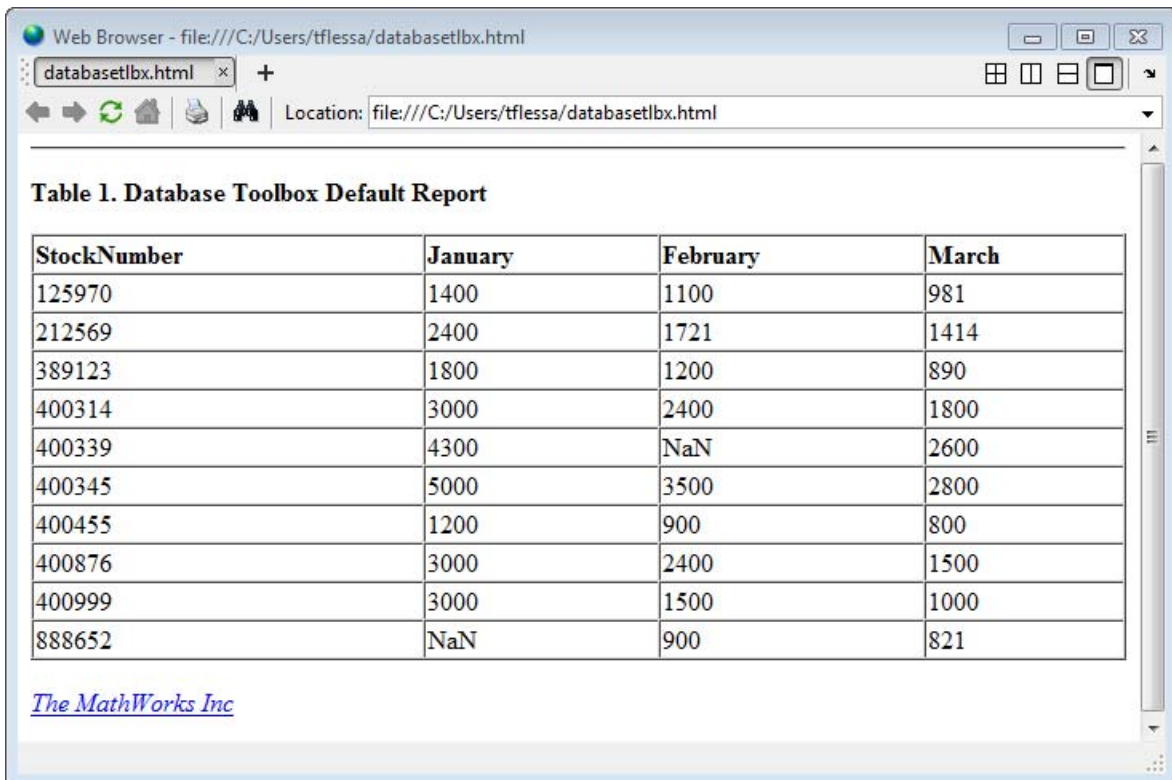


- 5** Field names do not automatically display as column headers in the report. To display the field names:
  - a** Modify the workspace variable A as follows:

```
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.



Web Browser - file:///C:/Users/tflessa/databasetlbx.html

databasetlbx.html x +

Location: file:///C:/Users/tflessa/databasetlbx.html

**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 more information 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...

“Retrieving All Occurrences vs. Unique Occurrences of Data” on page 3-27

“Retrieving Data That Meets Specified Criteria” on page 3-29

“Grouping Statements” on page 3-32

“Displaying Results in a Specified Order” on page 3-36

“Using Having Clauses to Refine Group by Results” on page 3-39

“Creating Subqueries for Values from Multiple Tables” on page 3-42

“Creating Queries That Include Results from Multiple Tables” on page 3-47

“Additional Advanced Query Options” on page 3-50

---

**Note** For more information 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 MATLAB 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]
```



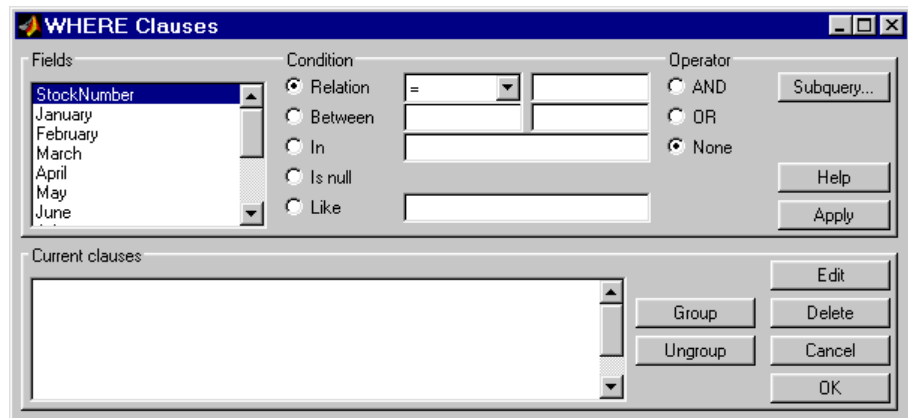
The value 3000 appears three times in All, but appears only once in Distinct.

## 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:

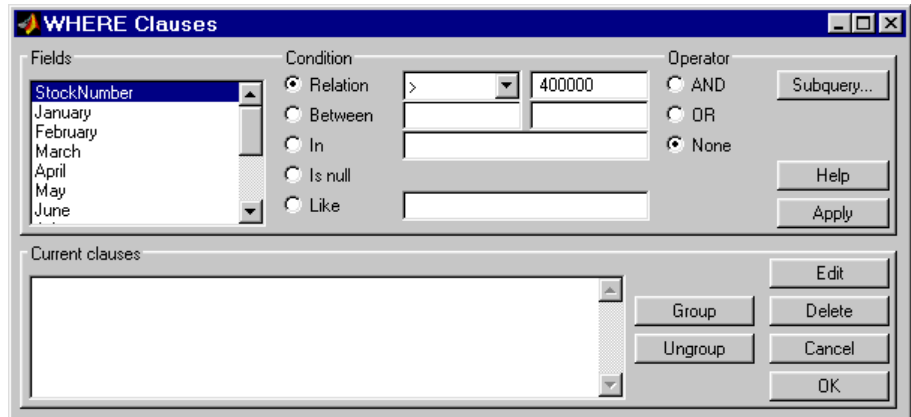
- 1 Load `basic.qry`.
- 2 Set the **Data return format** preference to `cellarray`.
- 3 Set **Read NULL numbers as** to `NaN`.
- 4 In **Advanced query options**, click **Where**.

The WHERE Clauses dialog box appears.



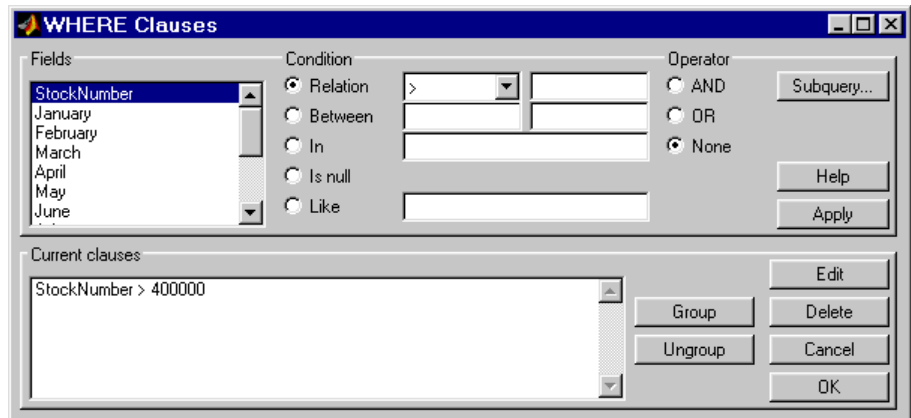
- 5 In **Fields**, select the field whose values you want to restrict, `StockNumber`.
- 6 In **Condition**, specify that `StockNumber` must be greater than 400000.
  - a Select **Relation**.
  - b In the drop-down list to the right of **Relation**, select `>`.
  - c In the field to the right of the drop-down 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]
```

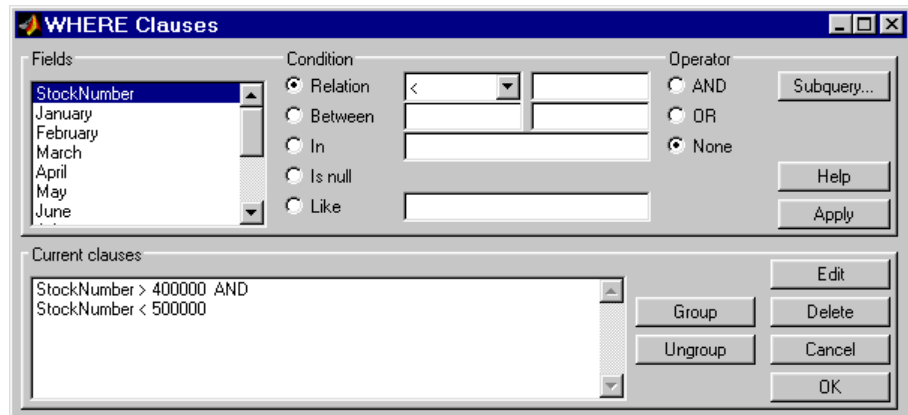
**12** Save this query as basic\_where.qry.

## 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:

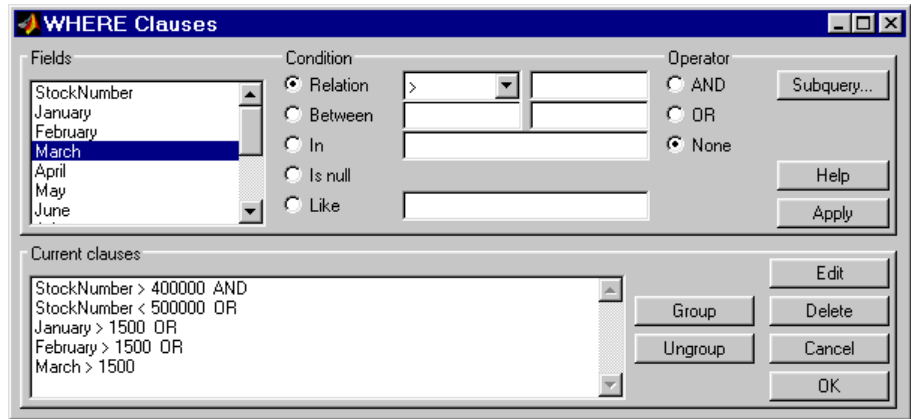
**1** Click **Where** in VQB. The WHERE Clauses dialog box appears.



**2** Modify the query to retrieve data if sales in January, February, or March exceed 1500 units.

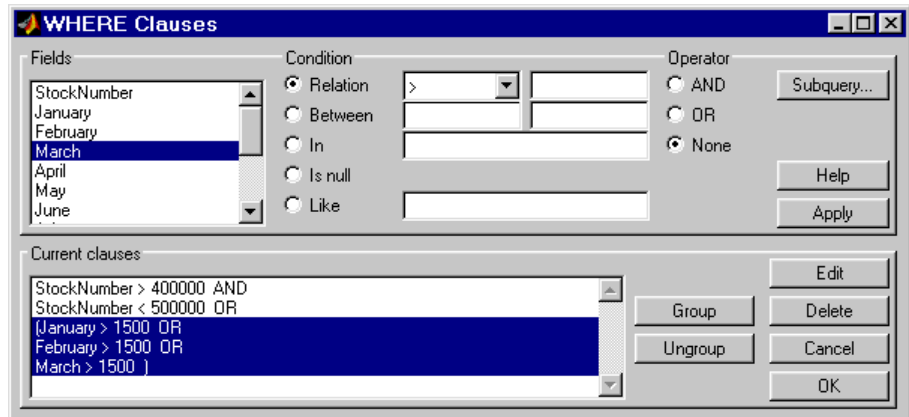
- a In **Current clauses**, select StockNumber < 500000 and click **Edit**.
- b For **Operator**, select OR and click **Apply**.
- c In **Fields**, select January. For **Relation**, select > and enter 1500 in its field. For **Operator**, select OR. Click **Apply**.
- d Repeat step c twice, specifying February and March in **Fields**.

The WHERE Clauses dialog box now looks as follows.



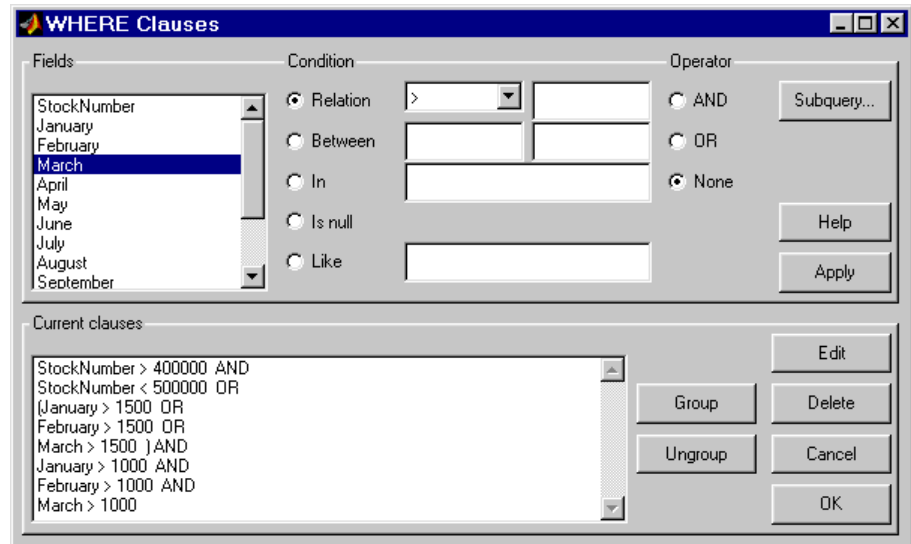
- 3 Group the criteria that require sales in each month to exceed 1500 units.
  - a In **Current clauses**, select the statement January > 1500 OR. Press **Shift**+click to select February > 1500 OR and March > 1500 also.
  - b Click **Group**.

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 MATLAB 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.

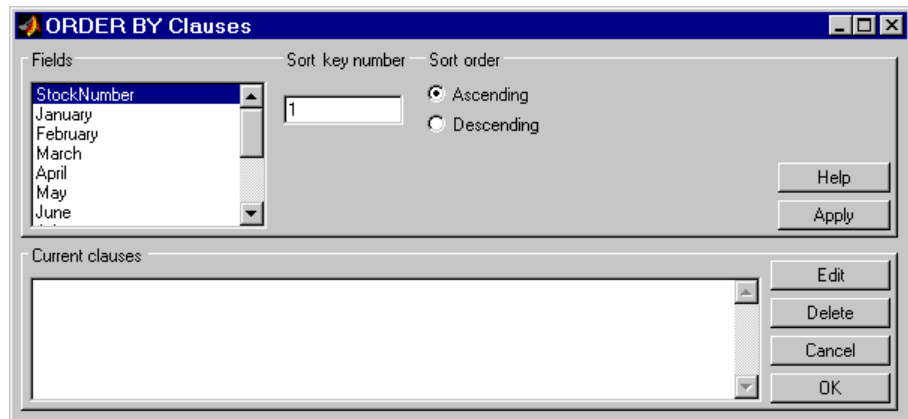
This example uses the `basic_where.qry` query you created in “Retrieving Data That Meets Specified Criteria” on page 3-29. The results of `basic_where.qry` are sorted so that January is the primary sort field, February the secondary, and March the last. Results for January and February appear in ascending 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 MATLAB 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.

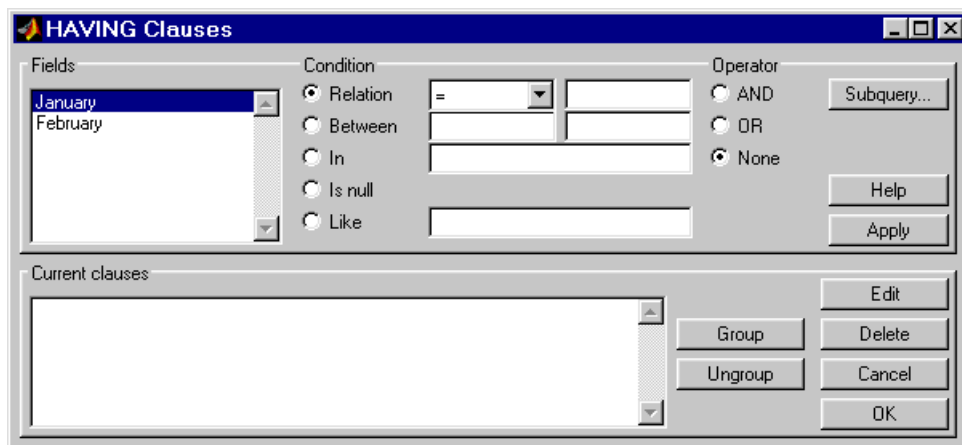
For items 400314 and 400876, **February sales** were 2400, so the third sort key, **March sales**, applies. **March sales** appear in descending order: 1800 and 1500, respectively.

## 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 3-29.
- 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:

**d** Select the clauses to group from **Current clauses**. Press **Ctrl+click** or **Shift+click** to select multiple clauses.

**e** 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 more information, see “Creating Subqueries for Values from Multiple Tables” on page 3-42.

**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**.

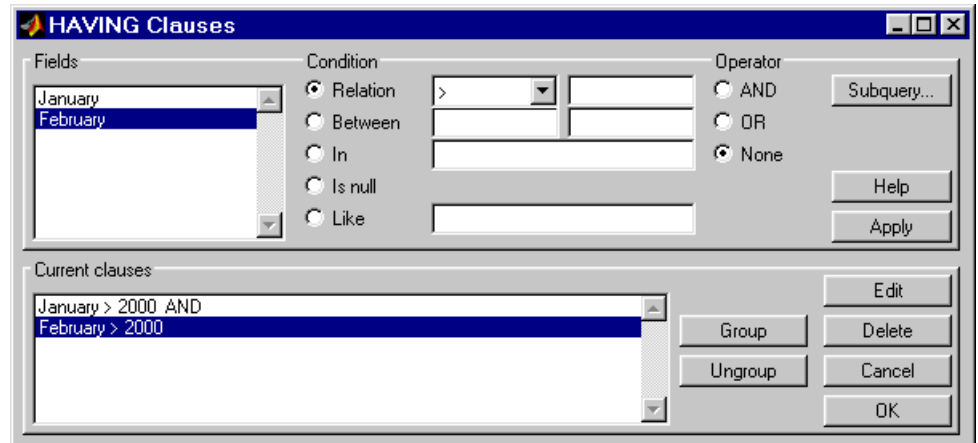
**d** Click **Apply**.

**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 3-36.

## 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 in “Saving Queries”.

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.

The screenshot shows the 'Subquery' dialog box with the following details:

- Title Bar:** Subquery
- Data source:** dbtoolboxdemo
- Tables:** inventoryTable, productTable, salesVolume, suppliers, productSales
- Fields:** (Empty)
- Subquery WHERE clauses:**
  - Fields:** (Empty list)
  - Condition:**
    - Relation =
    - Between
    - In
    - Is null
    - Like
  - Operator:**
    - AND
    - OR
    - None
  - Buttons:** Apply
- Current subquery WHERE clauses:**
  - Buttons:** Group, Edit, Ungroup, Delete
- SQL subquery statement:** (Empty text area)
- Bottom Buttons:** Cancel, Help, OK

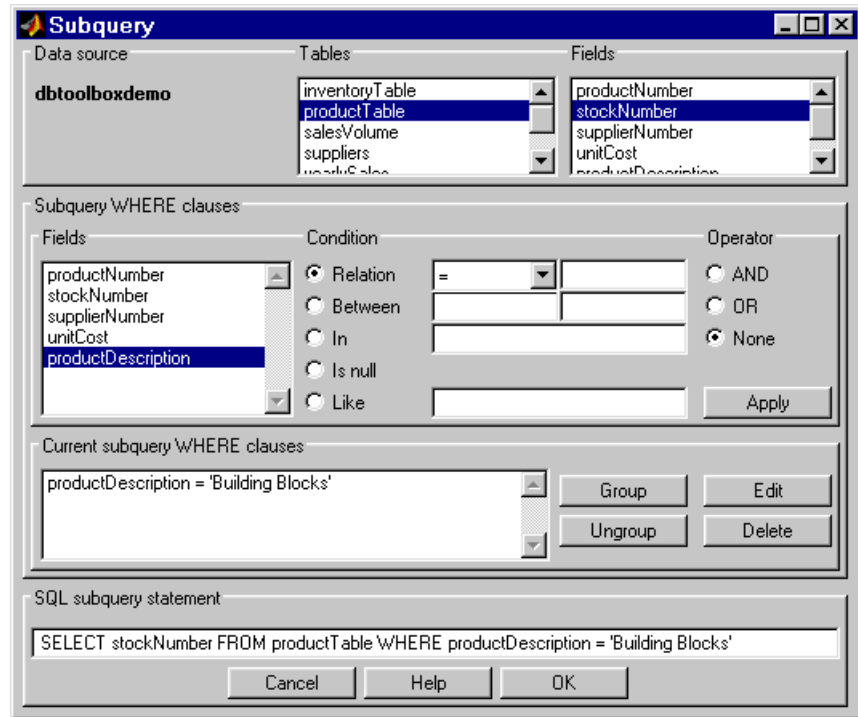
- 5 In **Tables**, select **productTable**, which includes the association between the stock number and the product description. The fields in that table appear.
- 6 In **Fields**, select **stockNumber**, the field that is common to this table and the table from which you are retrieving results.

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**.

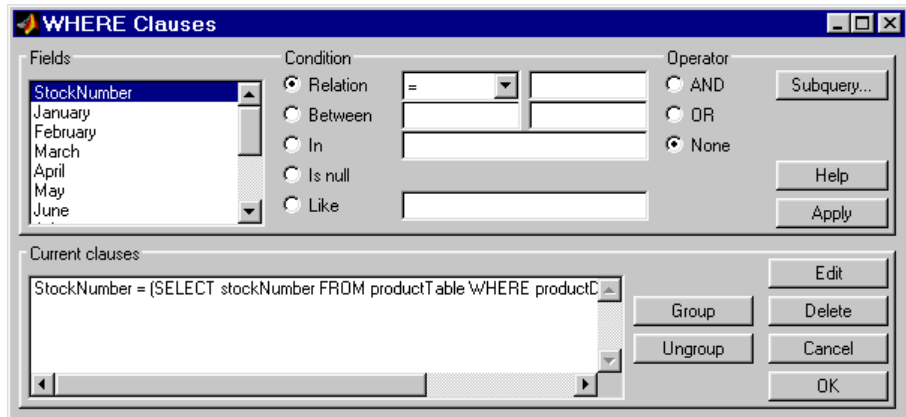




8 Click **OK** to close the Subquery dialog box.

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 MATLAB 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 MATLAB 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 3-42, 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.

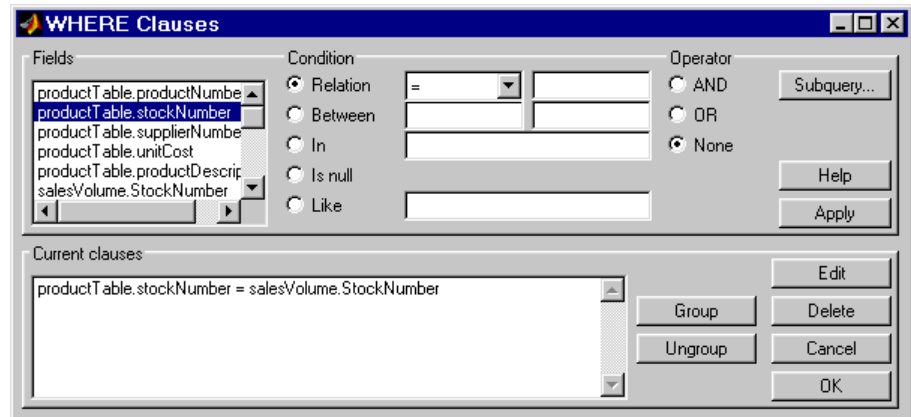
In **Advanced query options**, click **Where** to associate fields from different tables. The **WHERE Clauses** dialog box appears.

**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 MATLAB 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 more information on 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 Invoice.
- 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 MATLAB 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.

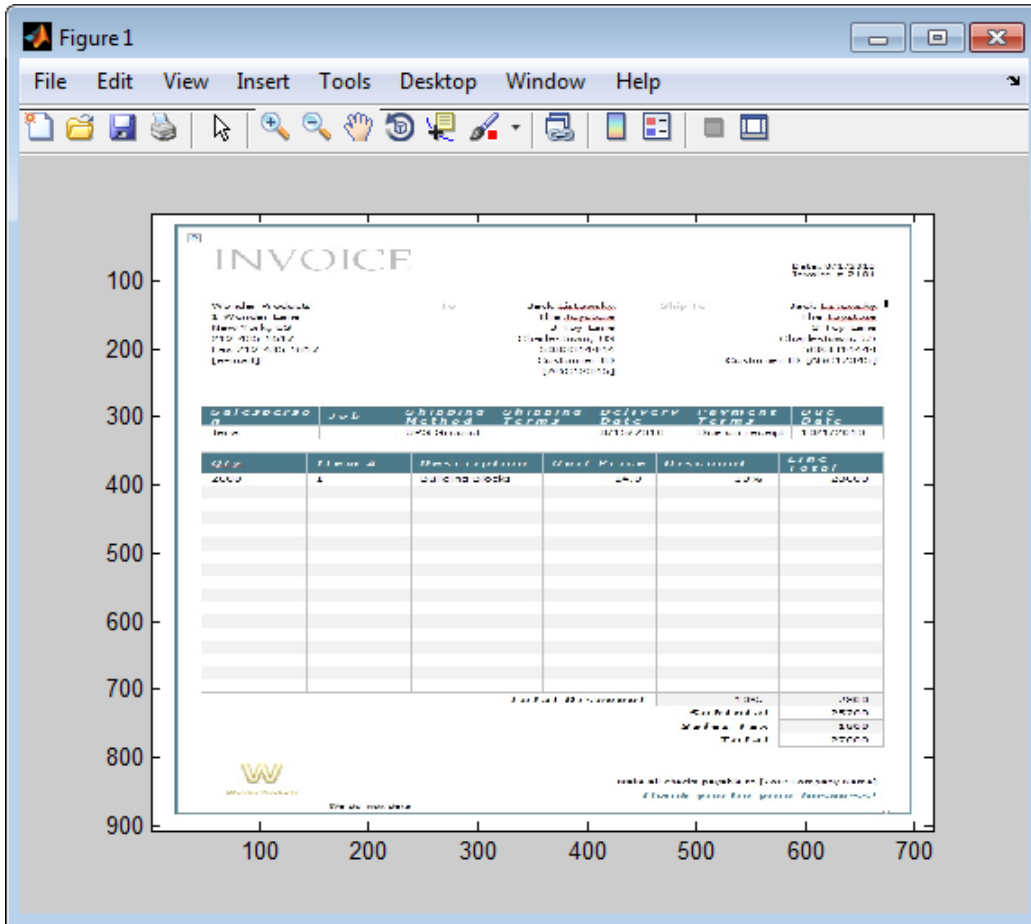
```
photo = A{1,2};
```

- 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 more information on 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...

“Importing BOOLEAN Data from Databases” on page 3-54

“Exporting BOOLEAN Data to Databases” on page 3-57

### Importing 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 MATLAB Command Window. 10 records are returned:

`D =`

```
[ 2101]    [0]
[ 3546]    [1]
[33116]    [1]
[34155]    [0]
[34267]    [1]
```

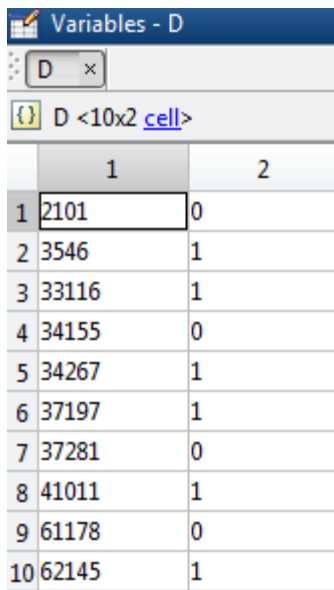
[37197] [1]  
 [37281] [0]  
 [41011] [1]  
 [61178] [0]  
 [62145] [1]

9 Compare these results to the data in Microsoft Access.

InvoiceNum	InvoiceDate	ProductNumber	Paid	Receipt
2101	8/1/2010	1	<input type="checkbox"/>	Bitmap Image
3546	3/1/2010	2	<input checked="" type="checkbox"/>	Bitmap Image
33116	5/15/2011	3	<input checked="" type="checkbox"/>	Bitmap Image
34155	7/12/2011	4	<input type="checkbox"/>	Bitmap Image
34267	7/22/2011	5	<input checked="" type="checkbox"/>	Bitmap Image
37197	9/3/2011	6	<input checked="" type="checkbox"/>	Bitmap Image
37281	9/21/2011	7	<input type="checkbox"/>	Bitmap Image
41011	12/12/2011	8	<input checked="" type="checkbox"/>	Bitmap Image
61178	1/15/2012	9	<input type="checkbox"/>	Bitmap Image
62145	1/23/2012	10	<input checked="" type="checkbox"/>	Bitmap Image

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.



The screenshot shows a window titled "Variables - D" with a tab labeled "D". Below the tab, there is a text field containing "D <10x2 cell>". Below this, a table is displayed with 10 rows and 2 columns. The first column contains integers from 1 to 10, and the second column contains binary values (0 or 1). The cell containing "2101" in the first row of the second column is highlighted with a black border.

	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			<input checked="" type="checkbox"/>		
456789			<input type="checkbox"/>		
2101	8/1/2010		1	<input type="checkbox"/>	Bitmap Image
3546	3/1/2010		2	<input checked="" type="checkbox"/>	Bitmap Image
33116	5/15/2011		3	<input checked="" type="checkbox"/>	Bitmap Image
34155	7/12/2011		4	<input type="checkbox"/>	Bitmap Image
34267	7/22/2011		5	<input checked="" type="checkbox"/>	Bitmap Image
37197	9/3/2011		6	<input checked="" type="checkbox"/>	Bitmap Image
37281	9/21/2011		7	<input type="checkbox"/>	Bitmap Image
41011	12/12/2011		8	<input checked="" type="checkbox"/>	Bitmap Image
61178	1/15/2012		9	<input type="checkbox"/>	Bitmap Image
62145	1/23/2012		10	<input checked="" type="checkbox"/>	Bitmap Image

## Saving Queries in Files

### In this section...

“About Generated Files” on page 3-59

“VQB Query Elements in Generated Files” on page 3-60

### 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 the string `'password'` as a placeholder. To run files to connect to databases that require passwords, substitute your password for the string `password` in the `database` statement.



## Using Database Explorer

### In this section...

- “About Database Explorer” on page 3-61
- “Workflow” on page 3-62
- “Configure Your Environment” on page 3-62
- “Database Connection Error Messages” on page 3-74
- “Set Database Preferences” on page 3-76
- “Display Data from a Single Database Table” on page 3-78
- “Join Data from Multiple Database Tables” on page 3-80
- “Define Query Criteria to Refine Results” on page 3-85
- “Query Rules Using the SQL Criteria Panel” on page 3-87
- “Query Example Using a Left Outer Join” on page 3-89
- “Work with Multiple Databases” on page 3-98
- “Import Data to the MATLAB Workspace” on page 3-98
- “Save Queries as SQL Code” on page 3-101
- “Generate MATLAB Code” on page 3-102

### About Database Explorer

`dexplore` starts Database Explorer, which is a Database Toolbox app for connecting to a database and importing data to the MATLAB workspace.

Database Explorer is an interactive app that lets you:

- 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 the MATLAB workspace.
- Save generated SQL queries.
- Generate MATLAB code.

### Workflow

The workflow for using Database Explorer is:

- 1 Set up an ODBC or JDBC connection to your database.
- 2 Connect to the database.
- 3 Select tables and columns of interest.
- 4 Use query criteria to fine-tune display results.
- 5 Select data and import it into the MATLAB workspace.
- 6 Use the Database Explorer tabbed interface to open multiple databases.
- 7 Save queries as an SQL script to run later.
- 8 Generate MATLAB code.

### Configure Your Environment

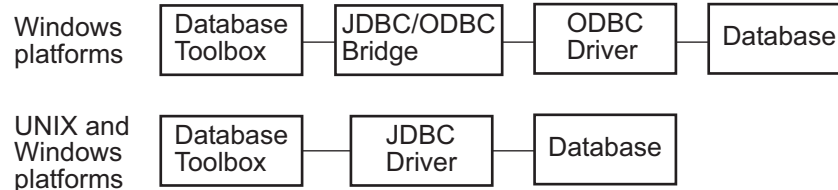
Before using Database Explorer to connect to a database, you must set up a *data source*. A data source consists of:

- 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 is automatically installed 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 ODBC/JDBC bridge is not used to access databases.

---

## Before You Begin

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

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

---

**Caution** If you have previously used Visual Query Builder (querybuilder) to access a JDBC data source, before starting Database Explorer for the first time, you must execute the following command:

```
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 3-64.

- If you are using a JDBC driver, see “Configure JDBC Data Sources” on page 3-69.

### Set Up the dbtoolboxdemo Data Source

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

To set up this data source:

- 1 Copy `tutorial.mdb` into a folder to which you have write access.
- 2 Confirm 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** You might need to convert the database to the version of Access you are currently running. For example, beginning in Microsoft Access 2007, you see the option to save as `*.accdb`. For more information, consult your database administrator.

---

### Configure ODBC Data Sources

When setting up a data source for use with an ODBC driver, the target database can be located 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. If you have a different configuration, you might need to modify these instructions. For more information, consult your database administrator.

- 1 Close open databases, including `tutorial.mdb` in the database program.
- 2 Start Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip and then selecting **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 dismiss the message. Otherwise, you are prompted to **Connect to a data source**. Click **Cancel** to dismiss this prompt.

- 3 Click the **Database Explorer** tab and then select **New > ODBC** to open the ODBC Data Source Administrator dialog box to define the ODBC data source.

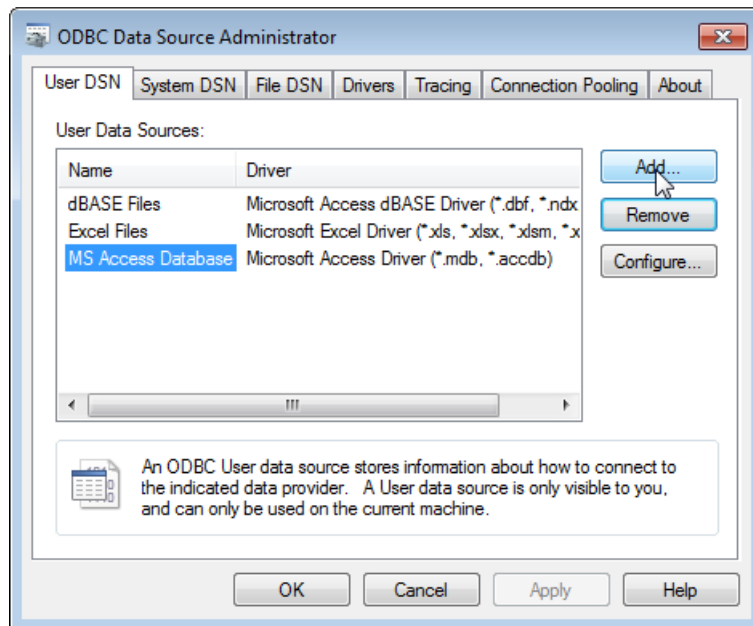
---

**Requirement** When using a 32-bit version of Microsoft Office, you must also use a 32-bit version of MATLAB to complete the remaining steps.

---

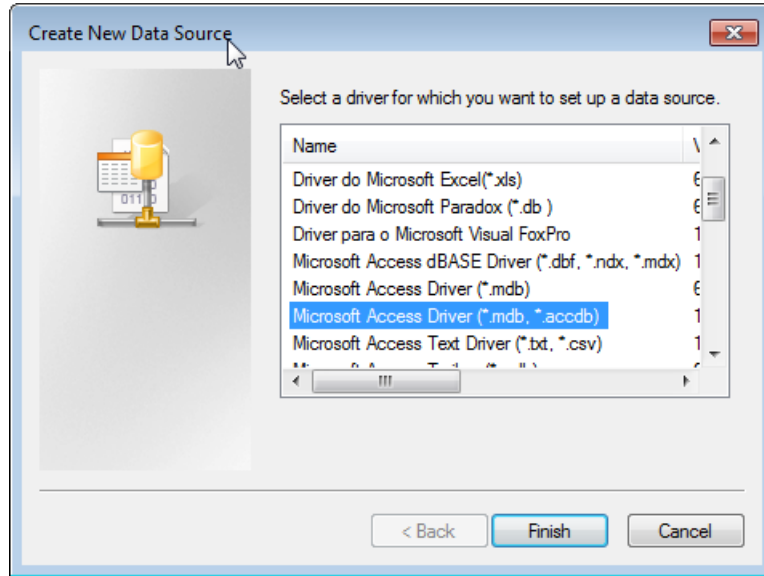
- 4 Click the **User DSN** tab.

- 5 Click **Add**.

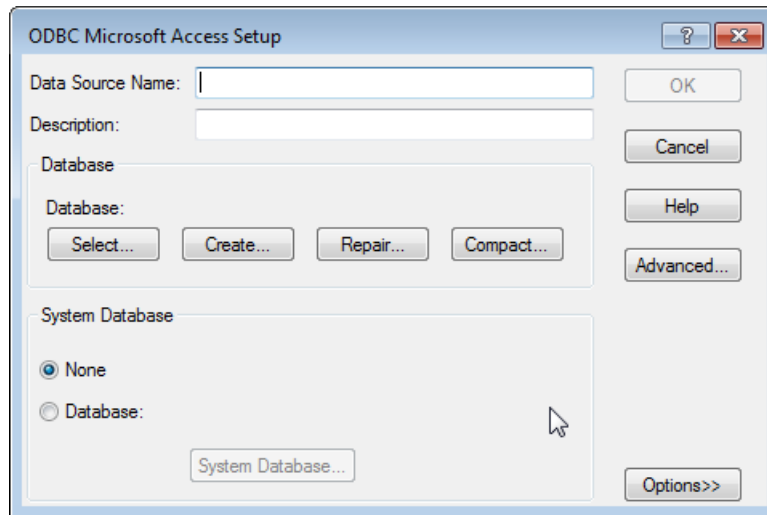


A list of installed ODBC drivers appears in the Create New Data Source dialog box.

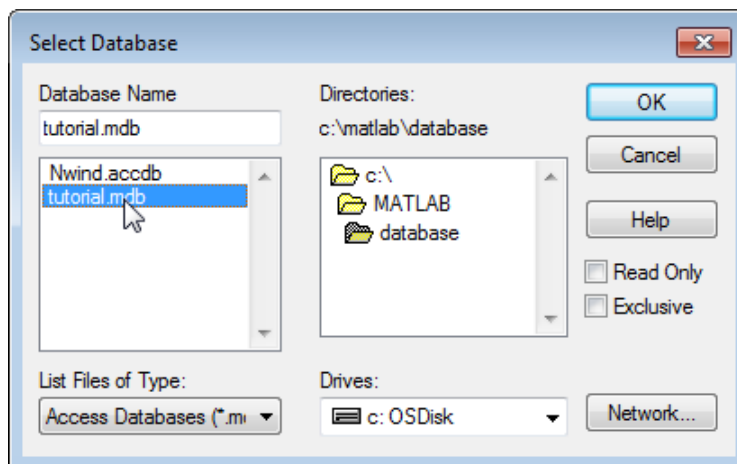
- 6 Select Microsoft Access Driver (\*.mdb, \*.accdb) and click **Finish**.



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



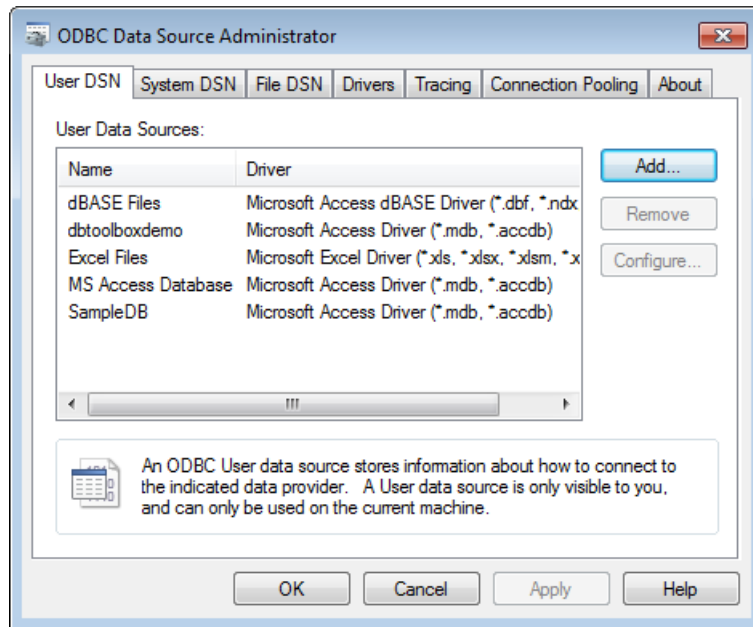
- 7 Enter dbtoolboxdemo as the data source name.
- 8 Enter tutorial database as the description.
- 9 Select the database for this data source to use. For some drivers, you can skip this step. If you are unsure about skipping this step, consult your database administrator.
  - a In the ODBC Microsoft Access Setup dialog box, click **Select**.



- b** Specify the database 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**. The Map Network Drive dialog box opens.
    - ii** Specify the folder containing the database you want to use.
    - iii** Click **Finish**.
  - d** Click **OK** to close the Select Database dialog box.
- 10** In the ODBC Microsoft Access Setup dialog box, click **OK**.
  - 11** Repeat steps 7 through 10 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.





**12** 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.

---

**Caution** If you have previously used Visual Query Builder (querybuilder) to access a JDBC data source, before starting Database Explorer for the first time, you must execute the following command:

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

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

---

- 2 Specify the location of the JDBC drivers file in the MATLAB Java class path by adding this file's path to `javaclasspath.txt` file. MATLAB loads the static class path at the start of each session. 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 named `javaclasspath.txt`. To view the location of the preferences folder, type:

```
prefdir
```

Each line in the file is the path name 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`, and `$jre_home`. You can also create a `javaclasspath.txt` file in your MATLAB startup folder. Classes specified in this file override classes specified in the `javaclasspath.txt` file in the preferences folder.

---

**Note** MATLAB reads the static class path only at startup. If you edit `javaclasspath.txt` or change your `.class` files while MATLAB is running, you must 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.

For more information, see “Bringing Java Classes into MATLAB Workspace”.

- 3 Close the open database, `tutorial.mdb`, in the database program.
- 4 Start Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip and then selecting **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 and then select **New > JDBC** to open the Create a New JDBC data source dialog box.

The screenshot shows the 'Create a New JDBC data source' dialog box. It has a title bar with a close button. The dialog is divided into two main sections: 'Data Source Details' and 'Connection Parameters'.  
 In the 'Data Source Details' section:  
 - 'Data Source Name' is a dropdown menu.  
 - 'Vendor' is a list box with 'MICROSOFT SQL SERVER' selected. Other visible options are MYSQL, ORACLE, and POSTGRESQL.  
 In the 'Connection Parameters' section:  
 - 'Server Name' is a text field containing 'localhost'.  
 - 'Port Number' is a text field containing '1433'.  
 - 'Authentication Type' is a dropdown menu set to 'Server'.  
 - 'Username', 'Password', and 'Database' are empty text fields.  
 At the bottom of the dialog, there is an information icon and a message: 'JDBC driver file was not found on MATLAB Java classpath'. Below this message are three buttons: 'Test', 'Save', and 'Delete'.

- 6 Use the following table to set up JDBC drivers for use with Database Explorer.
- a Using 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 used in these fields, see “JDBC Driver Name and Database Connection URL” on page 5-36 on the database function reference page.

<b>Field</b>	<b>Description</b>
<b>Data Source Name</b>	The name you assign to the data source. For some databases, the <b>Name</b> must match the name of the database as recognized by the machine it runs on.
<b>Vendor</b>	<p>The vendor's name for the data source. When using <b>Other</b>:</p> <ul style="list-style-type: none"> <li>• <b>Driver</b> — The JDBC driver name (sometimes referred to as the class that implements the Java SQL driver for your database).</li> <li>• <b>URL</b> — The JDBC URL object, of the form <code>jdbc:subprotocol:subname.subprotocol</code> is a database type. <code>subname</code> can contain other information used by <b>Driver</b>, such as the location of the database and/or a port number. It can take the form <code>//hostname:port/databasename</code>.</li> </ul> <hr/> <p><b>Note</b> When using <b>Other</b> as the <b>Vendor</b>, your driver manufacturer's documentation specifies the <b>Driver</b> and <b>URL</b> formats. You might need to consult your database system administrator for this information.</p> <hr/>
<b>Server Name</b>	Server name.
<b>Port Number</b>	Server port number.
<b>Authentication Type</b>	(Microsoft SQL Server only) Server or Windows authentication.
<b>Driver Type</b>	(Oracle only) Driver type is <b>thin</b> or <b>oci</b> .
<b>Username</b>	User name to access the database.
<b>Password</b>	Password.
<b>Database</b>	Database name.

- b** In the Create a New JDBC data source dialog box, click **Save**.
- c** If this is the first time 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.

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** Test the connection by clicking **Test**.

If your database requires a user name and password, a dialog box prompting you to supply them opens. 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 created 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 Data Source

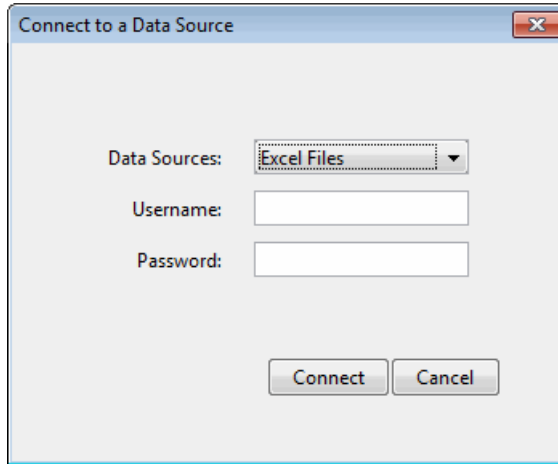
After configuring your ODBC or JDBC data sources, use Database Explorer to connect to the database.

- 1** Start Database Explorer by clicking the **Apps** tab on the MATLAB Toolstrip and then selecting **Database Explorer** from the **Database Connectivity and Reporting** section in the apps gallery. Alternatively, at the command line, enter:

```
dexplore
```

- 2** Select your data source from the Connect to a Data Source dialog box or click **Cancel** and then click the **Database Explorer** tab and then click **Connect** to select your data source.

- 3** Select your data source from the **Data Sources** list and enter your user name and password.



## Database Connection Error Messages

### Connection Error Messages and Probable Causes

Vendor	Error Message	Probable Causes
All	Unable to find JDBC driver.	<ul style="list-style-type: none"> <li>• Path to the JDBC driver jar file is not on the static or dynamic class path.</li> <li>• Incorrect driver name provided while using the 'driver' and 'url' syntax.</li> </ul>
All	[Microsoft][ODBC Driver Manager] The specified DSN contains an architecture mismatch between Driver and Application	You tried to open a 32-bit application when running MATLAB in 64-bit mode. Restart MATLAB to run in 32-bit mode using the command <code>matlab win32</code> .

**Connection Error Messages and Probable Causes (Continued)**

<b>Vendor</b>	<b>Error Message</b>	<b>Probable Causes</b>
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. Microsoft SQL Server uses a dynamic port for JDBC and the value should be verified using Microsoft SQL Server Configuration Manager.
Microsoft SQL Server	This driver is not configured for integrated authentication.	The Microsoft SQL Server Windows authentication library is not added to librarypath.txt. For more information, see the database example for Microsoft SQL Server Authenticated Database Connection.
Microsoft SQL Server	Invalid string or buffer length.	64-bit ODBC driver error. Use a JDBC driver or the native ODBC interface instead.
MySQL	Access denied for user 'user'@'machinename' (using password: YES)	Incorrect user name and password combination.
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.
MySQL	Unknown database 'databasename'.	Provided database name is incorrect.

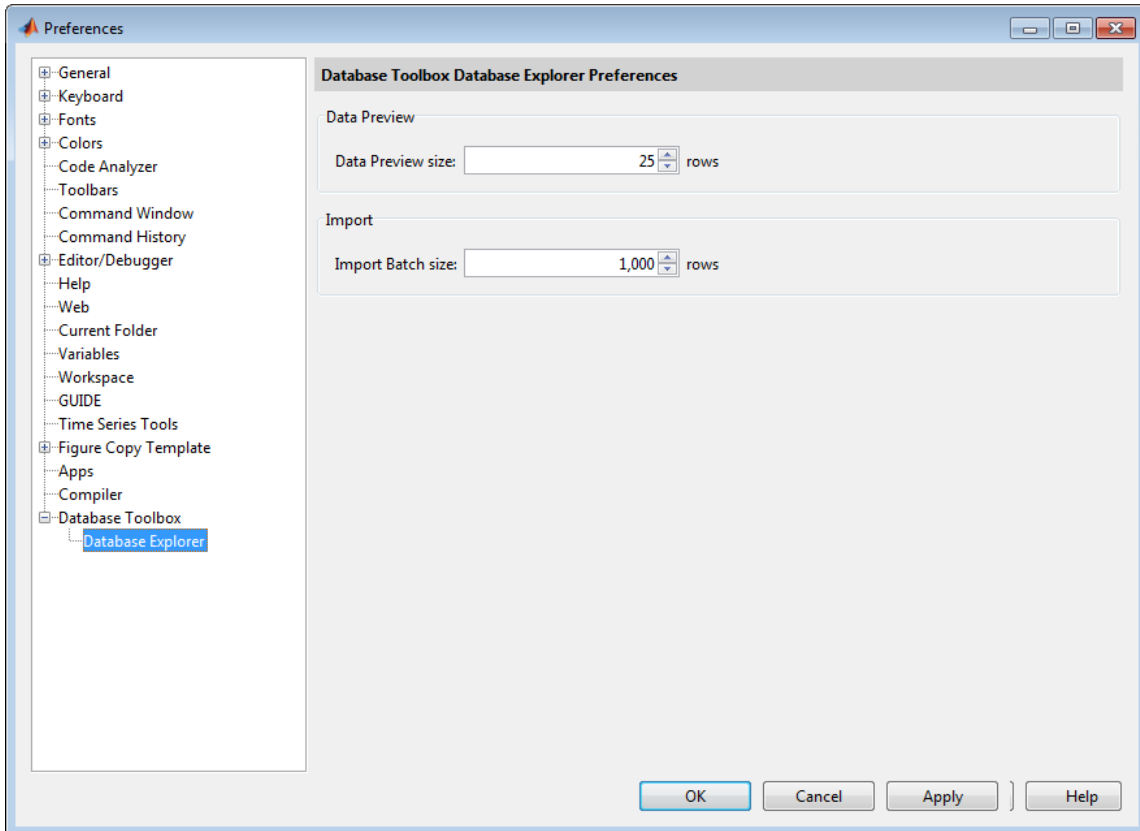
**Connection Error Messages and Probable Causes (Continued)**

Vendor	Error Message	Probable Causes
Oracle	<p>Error when connecting to Oracle oci8 database using JDBC driver:</p> <pre>Error using com.mathworks.toolbox.database.databaseConnector.openConnection@oracle Java exception occurred: java.lang.UnsatisfiedLinkError: no ocijdbc11 in java.library.pathat java.lang.ClassLoader.loadLibrary(Unknown Source)at java.lang.Runtime.loadLibrary0.....</pre>	<p>MATLAB cannot find the Oracle DLL that the oci8 drivers need. To correct the problem, add the path for the location of the Oracle DLLs to \$MATLAB/toolbox/local/librarypath.txt.</p>
Oracle	<p>Invalid Oracle URL specified:</p> <p>OracleDataSource.makeURL</p>	<p>DriverType parameter is not specified.</p>
Oracle	<p>The Network Adapter could not establish the connection.</p>	<p>Either Server or Portnumber is not specified or has an incorrect value.</p>

**Set Database Preferences**

- 1 Select **Preferences** from the Data Explorer Toolstrip to open the Database Toolbox 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.

<b>Preference</b>	<b>Allowable Values</b>	<b>Description</b>
<b>Data Preview size:</b>	5 to 10,000 rows	The number of rows you see in the <b>Data Preview</b> pane of Database Explorer.
<b>Import batch size:</b>	1,000 to 1,000,000 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 more information, see “Preference Settings for Large Data Import” on page 3-10.

From Database Toolbox Database Explorer Preferences dialog box, select **Database Toolbox** to manage additional preferences for Database Toolbox. For more information, see “Working with Preferences” on page 3-6. Alternatively, you can use `setdbprefs` to specify preferences for the retrieved data.

**3** Click **OK**.

## Display Data from a Single Database Table

After connecting to your database, you can display data in database tables in the **Data Preview** pane.

**1** Display data in the **Data Preview** pane by opening the database table of interest in the **Database Browser** pane. When a database table is selected in the **Database Browser** pane, it is highlighted and there is a corresponding entry in the **SQL Criteria** panel on the Database Explorer Toolstrip. The **SQL Criteria** panel is where you enter query conditions for the selected table.

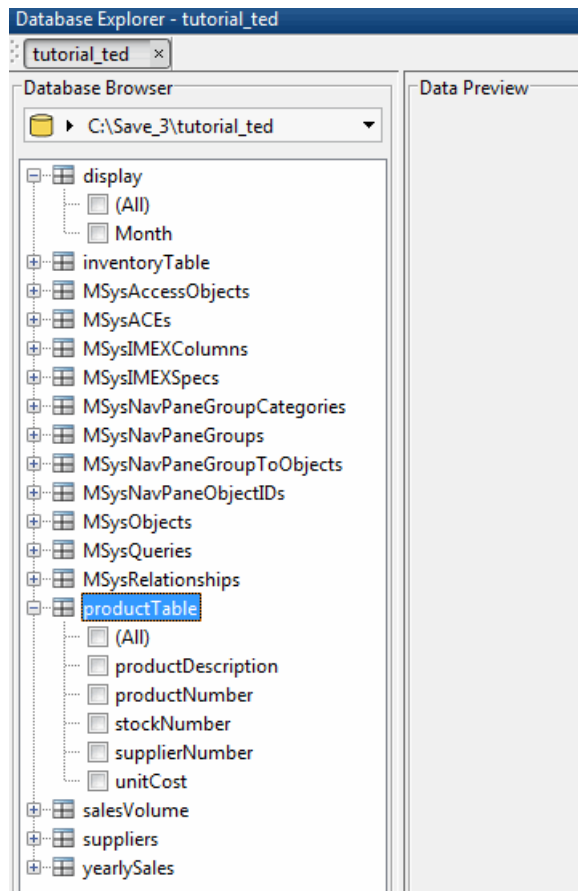
For any given table, you can select the table information any of three ways:

- Click to highlight the database table name. This 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.

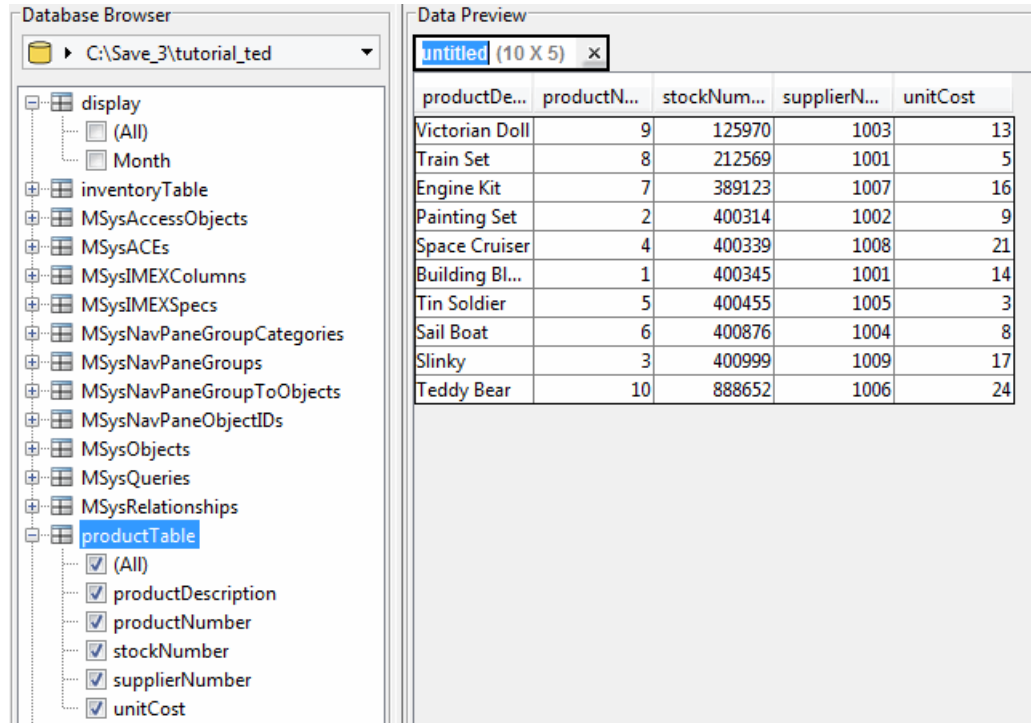
---

**Note** The order of the columns in the **Data Preview** pane matches the order in which you select them in the **Database Browser** pane.

---



- 2 Select **(All)** to choose all database columns or select check boxes for specific table columns.



- 3 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 actually 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**.

## Join Data from Multiple Database Tables

After connecting to your database, you can display data from database tables in the **Data Preview** pane.

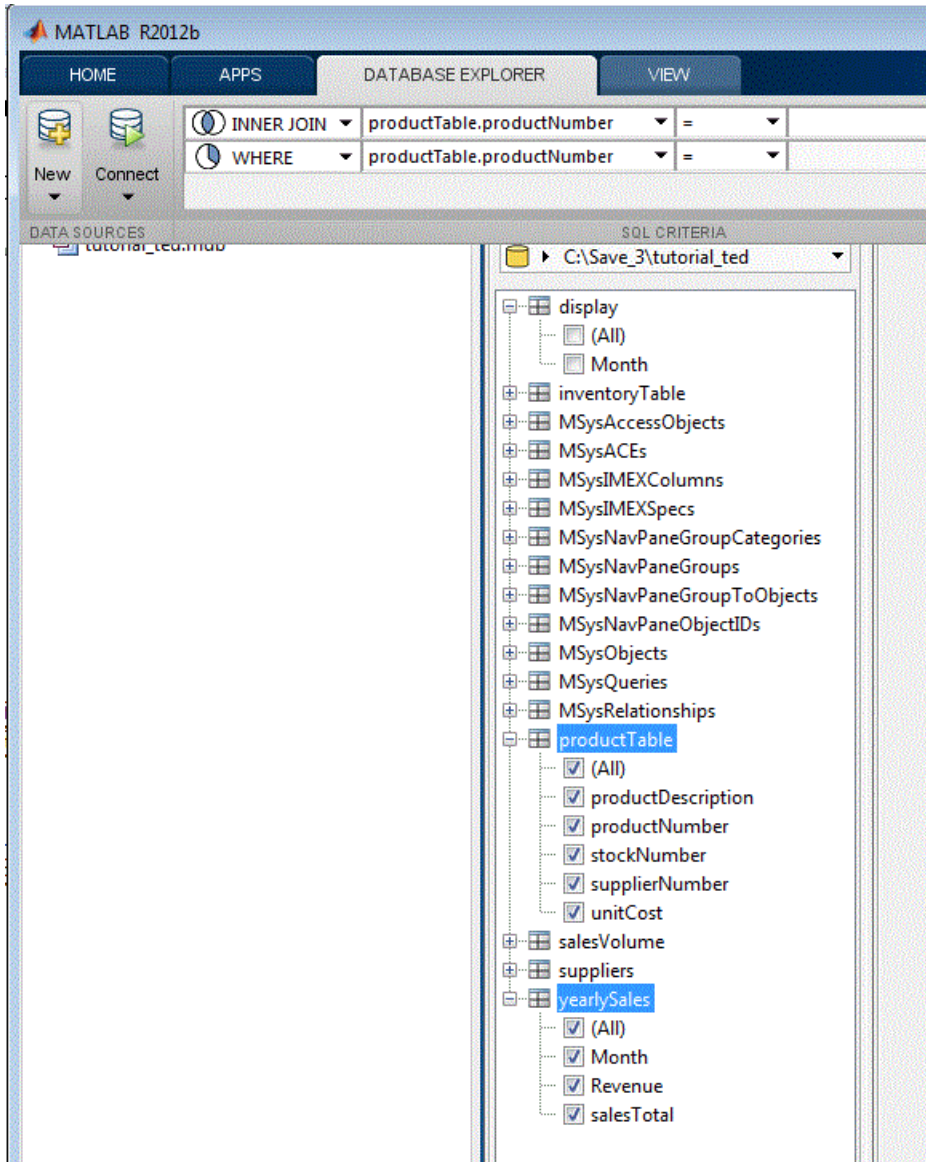
- 1 Display data in the **Data Preview** pane by opening the desired database table in the **Database Browser** pane. The **SQL Criteria** panel on the Database Explorer Toolstrip is updated.

The screenshot shows the Microsoft Access Database Explorer interface. The top ribbon includes 'HOME', 'APPS', 'DATABASE EXPLORER', and 'VIEW'. Below the ribbon, there are 'New' and 'Connect' buttons. The main area is divided into three panes:

- DATA SOURCES:** Shows a connection to 'tutorial\_ted.mdb'.
- SQL CRITERIA:** Displays a tree view of the database structure. The 'productTable' is selected, and its fields are listed with checkboxes:
  - (All)
  - productDescription
  - productNumber
  - stockNumber
  - supplierNumber
  - unitCost
- IMPORTED DATA:** Shows a table named 'untitled (10 X 5)' with the following data:
 

productDe...	productN...	stockNum...	supplierN...	t
Victorian Doll	9	125970	1003	
Train Set	8	212569	1001	
Engine Kit	7	389123	1007	
Painting Set	2	400314	1002	
Space Cruiser	4	400339	1008	
Building Bl...	1	400345	1001	
Tin Soldier	5	400455	1005	
Sail Boat	6	400876	1004	
Slinky	3	400999	1009	
Teddy Bear	10	888652	1006	

- 2** When you select additional tables in the **Database Browser** pane, the **SQL Criteria** panel is updated.



3 Display the contents for the selected table using the **SQL Criteria** panel to define a join of the selected tables. Click the drop-down lists to specify

which table column to join the selected tables. The join results appear in the **Data Preview** pane.

### 3 Using Visual Query Builder

The screenshot displays the Visual Query Builder interface in Microsoft Access. The top ribbon includes HOME, APPS, DATABASE EXPLORER, and VIEW. The query criteria are set to an INNER JOIN between productTable.stockNumber and salesVolume.StockNumber. The left pane shows the database structure with productTable and salesVolume tables expanded. The right pane shows the results grid with 10 rows of data.

productDe...	productN...	StockNum...	supplierN...
Victorian Doll	9	125970	1003
Train Set	8	212569	1001
Engine Kit	7	389123	1007
Painting Set	2	400314	1002
Space Cruiser	4	400339	1008
Building Bl...	1	400345	1001
Tin Soldier	5	400455	1005
Sail Boat	6	400876	1004
Slinky	3	400999	1009
Teddy Bear	10	888652	1006



## Define Query Criteria to Refine Results

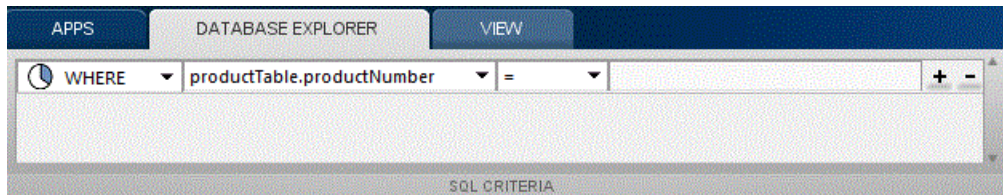
Database Browser selections and SQL criteria work together.

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 drop-down controls to define SQL query conditions. You can create SQL query conditions that span multiple rows in the **SQL Criteria** panel.

---

**Note** When the right side of a query condition is a custom value that you enter in the text box, you must press the **Enter** or **Tab** key for the query condition to take effect.

---



Each row in the **SQL Criteria** panel has four columns to define your SQL query.

Column 1	Column 2	Column 3	Column 4
<p>Column 1 defines the SQL condition type where supported values are:</p> <ul style="list-style-type: none"> <li>• <b>INNER JOIN</b></li> <li>• <b>LEFT JOIN</b></li> <li>• <b>RIGHT JOIN</b></li> <li>• <b>FULL JOIN</b></li> <li>• <b>WHERE</b></li> <li>• <b>ORDER BY</b></li> <li>• <b>AND</b></li> <li>• <b>OR</b></li> </ul>	<p>Column 2 defines the column names for all of the tables selected in the <b>Database Browser</b> pane.</p>	<p>Column 3 defines the mathematical operator for each row of SQL statements where supported values are:</p> <ul style="list-style-type: none"> <li>• <b>=</b></li> <li>• <b>!=</b></li> <li>• <b>&gt;</b></li> <li>• <b>&lt;</b></li> <li>• <b>&lt;=</b></li> <li>• <b>&gt;=</b></li> <li>• <b>LIKE</b></li> <li>• <b>NOT LIKE</b></li> <li>• <b>IS</b></li> <li>• <b>IN</b></li> <li>• <b>NOT IN</b></li> <li>• <b>ASC</b></li> <li>• <b>DES</b></li> </ul>	<p>Depending on the preceding condition of the query statement, Column 4 displays column names for all of the tables selected in the <b>Database Browser</b> pane.</p>

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 + or you can remove a row by clicking -.

- If only one table is selected in the **Database Browser** pane, the available options for the first query condition are **WHERE** and **ORDER BY**.
- If two tables are selected 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**

- After you apply a condition for a row in the **SQL Criteria** panel by using the **Enter** or **Tab** keys, for every subsequent condition that you add, the first (leftmost) column contains only those query options that produce semantically correct SQL statements. For example, if the leftmost column of an applied condition contains an **ORDER BY** option, if you click **+** to add a new query option in a new row, the **ORDER BY** option from the previous row can only be followed by another **ORDER BY** option.

In addition, a **Join** option can only be followed by another **JOIN** or **WHERE** and a **JOIN** option cannot follow a **WHERE** or **ORDER BY** option.

- When defining a new query line in the **SQL Criteria** panel for any conditions other than a **JOIN**, the new SQL line does not take effect until you apply the new line. When you apply a condition, all preceding and succeeding conditions that are not applied are removed from the **SQL Criteria** panel. Similarly, if you click **-** to remove a query line, if that query line has been applied, all succeeding conditions are removed. If the query line has not yet been applied, then only that line is removed from the **SQL Criteria** panel.
- When using a **WHERE** SQL statement with a mathematical operator, to match a string, you must include the string value in ' ' to successfully

apply the condition. If you use the **LIKE** or **NOT LIKE** SQL operator to match a string, the ' ' are automatically added to the string value.

---

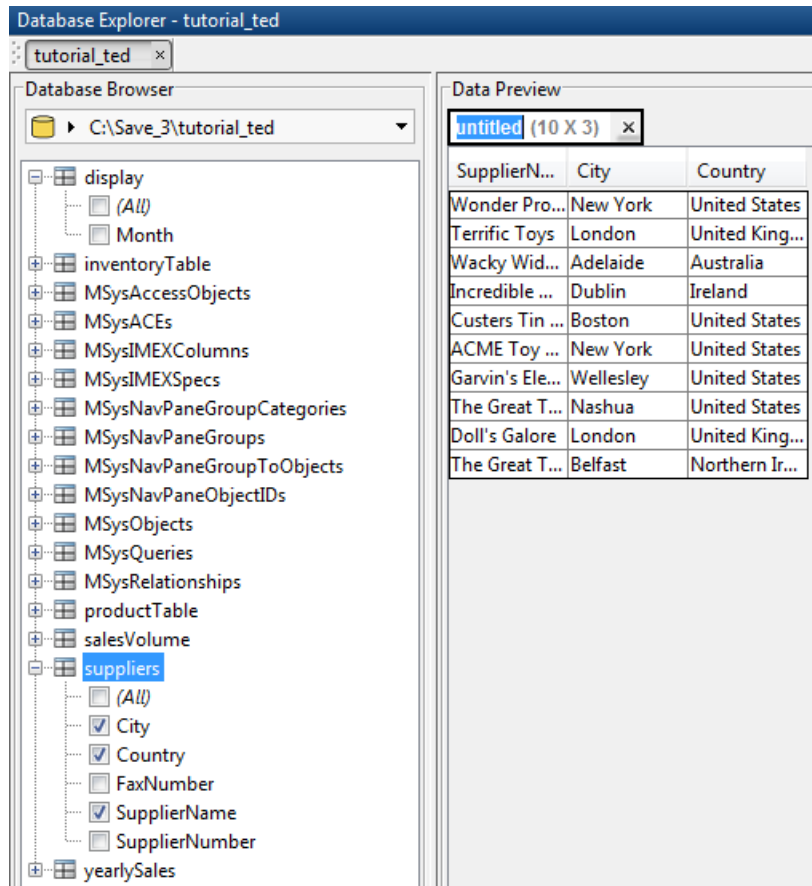
**Note** If you click + to add a new query condition between two previously entered conditions, the available query options do not always produce semantically correct SQL statements. In this case, you must ensure that your query options are semantically correct. For best results using the **SQL Criteria** panel, add and apply your conditions in sequence.

---

## Query Example Using a Left Outer Join

This example demonstrates how to use a query to obtain supplier and product information using a **LEFT JOIN**. To use this example, you must set up a data source for the `tutorial.mdb` database. For information on setting up this data source, see “Set Up the dbtoolboxdemo Data Source” on page 3-64.

- 1 Open `tutorial.mdb` in Database Explorer and expand the table `suppliers` and select the fields `SupplierName`, `City`, and `Country`.



- 2 Expand the table producttable and select the fields productDescription and unitCost. The **Data Preview** pane displays an info message prompting you to enter a join condition. Also, there are two empty conditions in the **SQL Criteria** panel on the Database Explorer Toolstrip.

The screenshot shows the Microsoft Access Database Explorer interface. At the top, there are tabs for PLOTS, APPS, DATABASE EXPLORER, and VIEW. Below these tabs is the SQL CRITERIA panel, which contains two conditions. The first condition is set to INNER JOIN, and the second is set to WHERE. Both conditions have 'productTable.productNumber' as the field and '=' as the operator. To the right of the SQL CRITERIA panel is a dropdown menu with options: Cell Array, Numeric, Structure, and Dataset. Below the SQL CRITERIA panel is the IMPORTED DATA panel, which includes buttons for Import and Preferences. The main area of the Database Explorer is split into two panes: Database Browser on the left and Data Preview on the right. The Database Browser pane shows a tree view of the database 'tutorial\_ted\_2' with various tables and fields. The 'productTable' and 'suppliers' tables are selected. The Data Preview pane is empty and contains a message: 'You have selected more than one table. Please enter a join condition for each table, or remove unwanted tables from the selection.'

- From the **SQL Criteria** panel, in the first (topmost) condition, 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, and 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. For the supplier named The Great Teddy Bear Company, notice that there is a null in the productDescription and a NaN for unitCost. This is because there is no product that is supplied by The Great Teddy Bear Company. If the condition type were **INNER JOIN** instead of **LEFT JOIN**, this row would not be present in the final result.



Database Explorer - tutorial\_ted

LEFT JOIN suppliers.SupplierNumber = productTable.supplierNumber

SQL CRITERIA

IMPORTED DATA

Import

Database Browser

C:\Save\_3\tutorial\_ted

- display
  - (All)
  - Month
- inventoryTable
- MSysAccessObjects
- MSysACEs
- MSysIMEXColumns
- MSysIMEXSpecs
- MSysNavPaneGroupCategories
- MSysNavPaneGroups
- MSysNavPaneGroupToObjects
- MSysNavPaneObjectIDs
- MSysObjects
- MSysQueries
- MSysRelationships
- productTable
  - (All)
  - productDescription
  - productNumber
  - stockNumber
  - supplierNumber
  - unitCost
- salesVolume
- suppliers
  - (All)
  - City
  - Country
  - FaxNumber
  - SupplierName
  - SupplierNumber
- yearlySales

Data Preview

Untitled (11 X 5)

SupplierName	City	Country	productDescri...	unitCost
Wonder Products	New York	United States	Building Blocks	14.0
Wonder Products	New York	United States	Train Set	5.0
Terrific Toys	London	United Kingdom	Painting Set	9.0
Wacky Widgets	Adelaide	Australia	Victorian Doll	13.0
Incredible Machines	Dublin	Ireland	Sail Boat	8.0
Custers Tin Soldiers	Boston	United States	Tin Soldier	3.0
ACME Toy Company	New York	United States	Teddy Bear	24.0
Garvin's Electrical Gizmos	Wellesley	United States	Engine Kit	16.0
The Great Train Company	Nashua	United States	Space Cruiser	21.0
Doll's Galore	London	United Kingdom	Slinky	17.0
The Great Teddy Bear Company	Belfast	Northern Ireland	null	NaN

- 4 From the **SQL Criteria** pane, click **+** at the end of the **LEFT JOIN** condition to add a new query condition. Change the first combo box to **WHERE**, the second to **suppliers.Country**, the third to **NOT LIKE**. In the last text box, type **United State** and then enter the new condition using the **Enter** or **Tab** key. The query results display in the **Data Preview** pane.

Database Explorer - tutorial\_ted

SQL CRITERIA

Operator	Field 1	Comparison	Field 2	Buttons
LEFT JOIN	suppliers.SupplierNumber	=	productTable.supplierNumber	+ -
WHERE	suppliers.Country	NOT LIKE	'United States'	+ -

Database Browser

C:\Save\_3\tutorial\_ted

- display
  - (All)
  - Month
- inventoryTable
- MSysAccessObjects
- MSysACEs
- MSysIMEXColumns
- MSysIMEXSpecs
- MSysNavPaneGroupCategories
- MSysNavPaneGroups
- MSysNavPaneGroupToObjects
- MSysNavPaneObjectIDs
- MSysObjects
- MSysQueries
- MSysRelationships
- productTable
  - (All)
  - productDescription
  - productNumber
  - stockNumber
  - supplierNumber
  - unitCost
- salesVolume
- suppliers
  - (All)
  - City
  - Country
  - FaxNumber
  - SupplierName
  - SupplierNumber
- yearlySales

Data Preview

untitled (5 X 5)

SupplierName	City	Country	productDesc...	unitCost
Terrific Toys	London	United Kingdom	Painting Set	9.0
Wacky Widgets	Adelaide	Australia	Victorian Doll	13.0
Incredible Machines	Dublin	Ireland	Sail Boat	8.0
Doll's Galore	London	United Kingdom	Slinky	17.0
The Great Teddy Bear Company	Belfast	Northern Ireland	null	NaN

- 5 Enter the variable name as **data** in the text box named **untitled** located above the table preview, and select **Import > Import** to import the data displayed in the **Data Preview** pane into MATLAB as a variable named **data**. For more information about using the MATLAB Variables editor, see “View, Edit, and Copy Variables”.

Database Explorer - tutorial\_ted

SQL CRITERIA

IMPORTED DATA

Database Explorer - tutorial\_ted

tutorial\_ted

Database Browser

C:\Save\_3\tutorial\_ted

- display
  - (All)
  - Month
- inventoryTable
- MSysAccessObjects
- MSysACEs
- MSysIMEXColumns
- MSysIMEXSpecs
- MSysNavPaneGroupCategories
- MSysNavPaneGroups
- MSysNavPaneGroupToObjects
- MSysNavPaneObjectIDs
- MSysObjects
- MSysQueries
- MSysRelationships
- productTable
  - (All)
  - productDescription
  - productNumber
  - stockNumber
  - supplierNumber
  - unitCost
- salesVolume
- suppliers
  - (All)
  - City
  - Country
  - FaxNumber
  - SupplierName
  - SupplierNumber
- yearlySales

Data Preview

data (5 X 5)

SupplierName	City	Country	productDesc...	unitCost
Terrific Toys	London	United Kingdom	Painting Set	9.0
Wacky Widgets	Adelaide	Australia	Victorian Doll	13.0
Incredible Machines	Dublin	Ireland	Sail Boat	8.0
Doll's Galore	London	United Kingdom	Slinky	17.0
The Great Teddy Bear Company	Belfast	Northern Ireland	null	NaN

## Work with Multiple Databases

- 1** If you have not defined the ODBC or JDBC connection for your new data source, click **Open** and select **ODBC** or **JDBC** and complete the associated dialog box. For more information, see “Configure ODBC Data Sources” on page 3-64 or “Configure JDBC Data Sources” on page 3-69.
- 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.

Note, you can only use Database Explorer to create SQL queries for a single database at a time.

In addition, you can work with a different catalog and schema on the same database server as the one connected to your current data source. To change to a different catalog and schema:

- Select the catalog/schema from the drop-down list in the address bar of the Database Browser. For a database system like Microsoft SQL Server that has a hierarchy of catalogs and schemas, make sure you choose the correct value for both in order to access data in your tables.

## Import Data to the MATLAB Workspace

- 1** Use the **Database Browser** pane to select data from a single table or use the **SQL Criteria** panel to create a query and display the results in the **Data Preview** pane.
- 2** Name the MATLAB variable by entering it in the **untitled** text box in the **Data Preview** pane.
- 3** Use the **Imported Data** panel to define the data structure for a MATLAB variable to store the data displayed in the **Data Preview** pane. Supported data structures are:
  - **Cell Array**
  - **Numeric**
  - **Structure**
  - **Table**

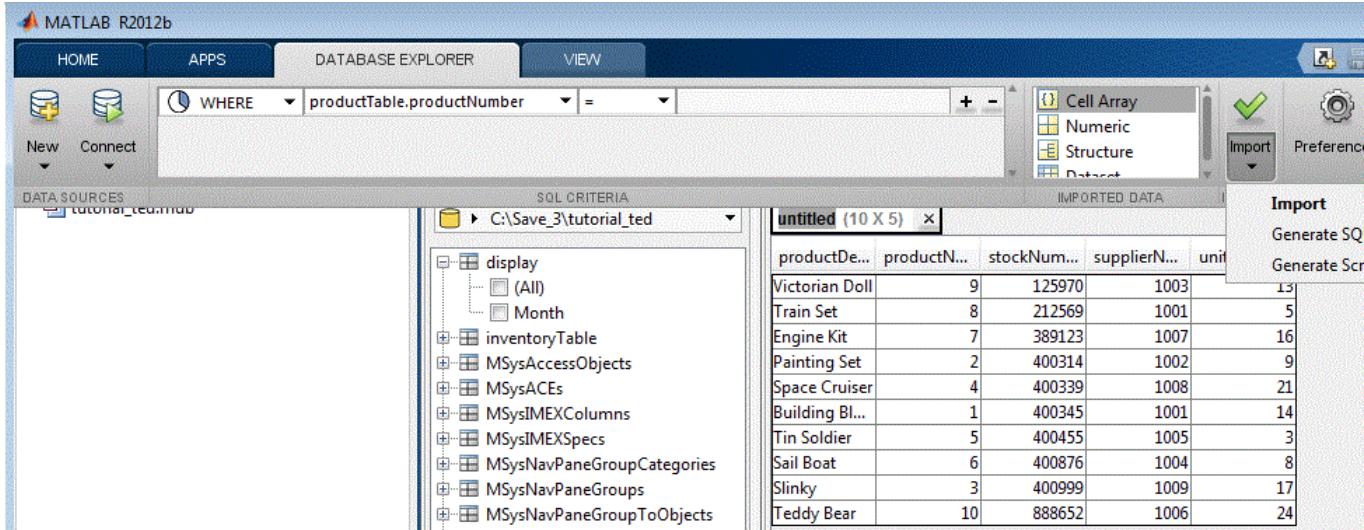
- **Dataset** (requires Statistics Toolbox)

The screenshot shows the Microsoft SQL Server Enterprise Manager interface. The 'display' folder is expanded to show the 'productTable'. The 'productTable' is selected, and its columns are visible in the 'Data Preview' pane. The columns are: productDe..., productN..., stockNum..., supplierN..., and unitCost. The data preview shows 10 rows of data.

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

- 4 Select **Import** > **Import** to import the data displayed in the **Data Preview** pane.

### 3 Using Visual Query Builder



The screenshot shows the MATLAB R2012b Database Explorer interface. The 'VIEW' tab is active, displaying a query result table. The query criteria are set to 'WHERE productTable.productNumber ='. The table has 10 rows and 5 columns: productDe..., productN..., stockNum..., supplierN..., and unit. The data is as follows:

productDe...	productN...	stockNum...	supplierN...	unit
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

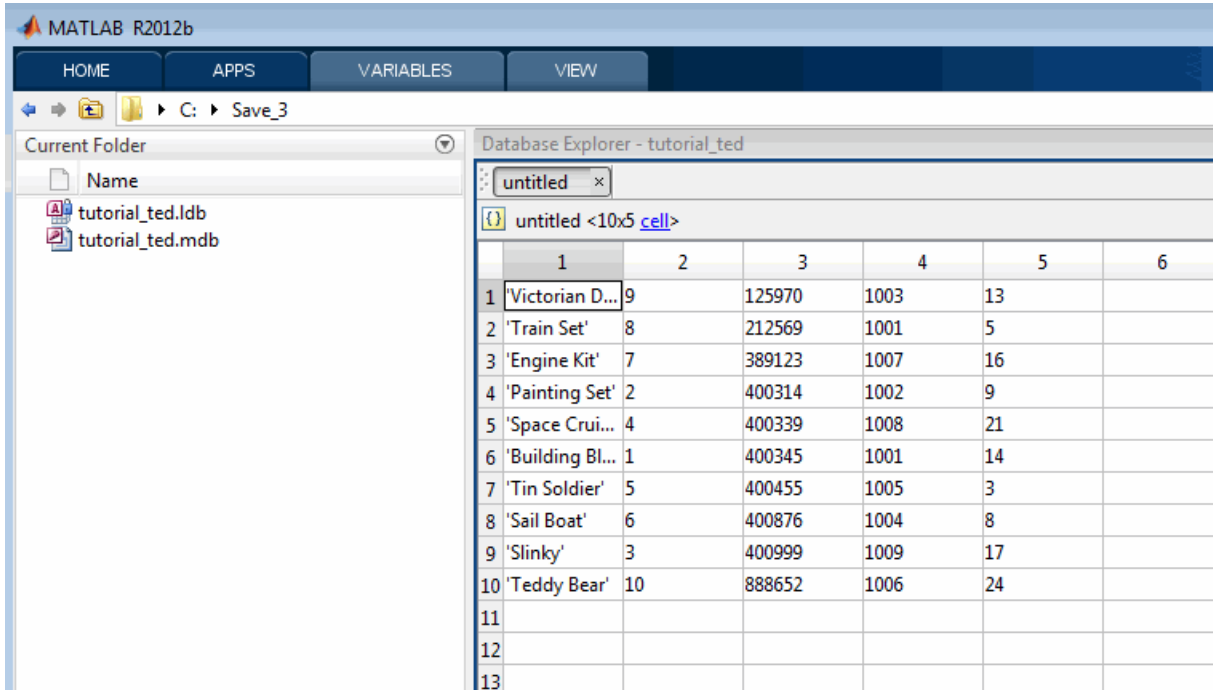
---

**Tip** 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**.

---



- (Optional) Display the imported data in the MATLAB workspace using the Variables editor. For more information about using the Variables editor, see “View, Edit, and Copy Variables”.



The screenshot shows the MATLAB R2012b interface with the Database Explorer pane open. The current folder is 'C:\Save\_3' and it contains two files: 'tutorial\_ted.ldb' and 'tutorial\_ted.mdb'. The Database Explorer pane displays a table with 6 columns and 13 rows. The data is as follows:

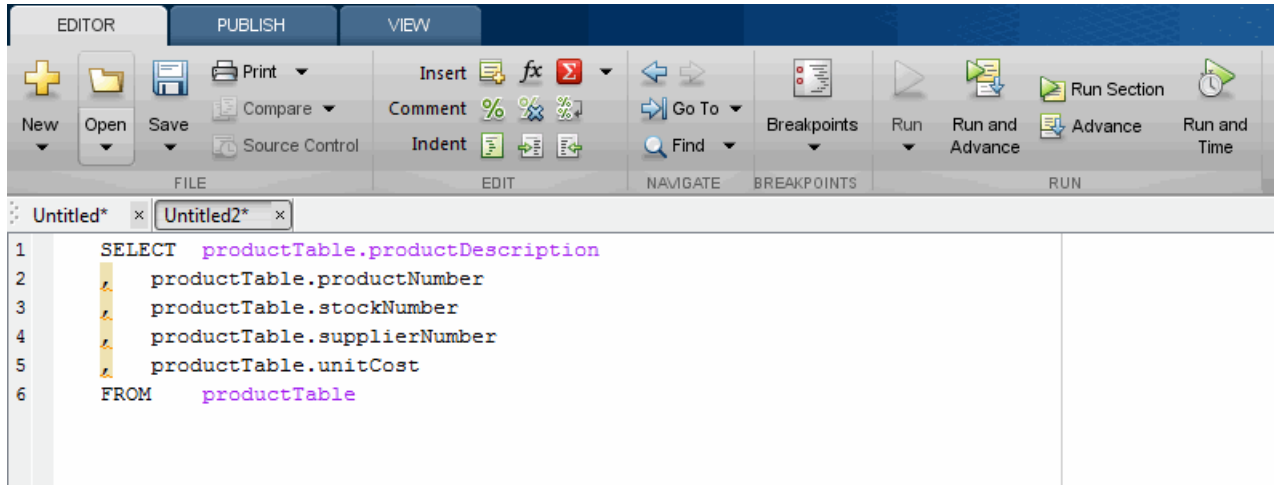
	1	2	3	4	5	6
1	'Victorian D...	9	125970	1003	13	
2	'Train Set'	8	212569	1001	5	
3	'Engine Kit'	7	389123	1007	16	
4	'Painting Set'	2	400314	1002	9	
5	'Space Crui...	4	400339	1008	21	
6	'Building Bl...	1	400345	1001	14	
7	'Tin Soldier'	5	400455	1005	3	
8	'Sail Boat'	6	400876	1004	8	
9	'Slinky'	3	400999	1009	17	
10	'Teddy Bear'	10	888652	1006	24	
11						
12						
13						

- (Optional) Use MATLAB functions to manipulate the data.

## Save Queries as SQL Code

You can save a Database Explorer query as SQL code.

- Use the **Database Browser** pane to select data from a single table or multiple tables. Then use the **SQL Criteria** panel to create queries and display the results in the **Data Preview** pane.
- After you have created a query using the **SQL Criteria** panel, click **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 manually rebuild a query using the **SQL Criteria** panel.

**4** 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 the steps for accessing data that you display in the **Data Preview** pane.

**1** Connect to a data source and then use the **Database Browser** pane to select data from a single table or 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.

The screenshot shows the MATLAB Editor window with a script titled 'Untitled3\*'. The script contains MATLAB code for connecting to a database, executing a query, and displaying the results. The code is as follows:

```

1  %Set preferences with setdbprefs.
2  setdbprefs('DataReturnFormat', 'cellarray');
3  setdbprefs('NullNumberRead', 'NaN');
4  setdbprefs('NullStringRead', 'null');
5
6
7  %Make connection to database. Note that the password has been omitted
8  %Using ODBC driver.
9  conn = database('tutorial_ted', 'admin', '');|
10
11 %Read data from database.
12 curs = exec(conn, ['SELECT productTable.productDescription'...
13     ', productTable.productNumber'...
14     ', productTable.stockNumber'...
15     ', productTable.supplierNumber'...
16     ', productTable.unitCost'...
17     ' FROM productTable ']);
18
19 curs = fetch(curs);
20 close(curs);
21
22 %Assign data to output variable
23 untitled = curs.Data;
24
25 %Close database connection.
26 close(conn);
27
28 %Clear variables
29 clear curs conn

```

- 3 Save the MATLAB code to a file. You can run this code file from the command line to connect to a data source and run a query.



# Using Database Toolbox Functions

---

- “Getting Started with Database Toolbox Functions” on page 4-2
- “Importing Data from Databases” on page 4-3
- “Viewing Information About Imported Data” on page 4-5
- “Exporting Data to New Record in Database” on page 4-8
- “Replacing Existing Database Data with Exported Data” on page 4-12
- “Exporting Multiple Records from the MATLAB Workspace” on page 4-14
- “Exporting Data Using the Bulk Insert Command” on page 4-18
- “Retrieving Image Data Types” on page 4-25
- “Working with Database Metadata” on page 4-27
- “Using Driver Functions” on page 4-34
- “About Database Toolbox Objects and Methods” on page 4-36
- “Using the exec Function” on page 4-39
- “Using the fetch Function” on page 4-42

# 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.

## Importing Data from Databases

This example demonstrates a sample workflow on a demonstration database called `dbtoolboxdemo`.

- 1 Before you connect to a database, set the maximum time that you want to allow the MATLAB software session to try to connect to a database to 5 seconds.

```
logintimeout(5)
```

---

**Note** If you are connecting to a database using a JDBC connection, you need to specify different function syntax in this step. For more information, see the `logintimeout` function reference page.

---

- 2 Use the `database` function to define a MATLAB variable, `conn`, to represent the returned connection object. Pass the following arguments to this function:
  - The name of the database, which is `dbtoolboxdemo` for this example
  - The username and password

```
conn = database('dbtoolboxdemo', 'username', 'password')
```

Enter `conn` at the command prompt to see the data.

---

**Note** If you are connecting to a database using a JDBC connection, you need to specify different syntax for the `database` function. Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see the `database` reference page.

---

- 3 Use `ping` to check that the database connection status is successful.
- 4 Use the `exec` function to open a cursor and execute an SQL statement. Pass the following arguments to `exec`:
  - `conn`, the name of the connection object

- `select productNumber from productTable`, a SQL statement that selects the `productNumber` column of data from the `productTable` table

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

The `exec` function returns the MATLAB variable `curs`.

- 5 The returned data contains strings, so you must convert it to a format that supports strings. Use `setdbprefs` to specify the format `cellarray`:

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

- 6 To stop working now and resume working on the next example at a later time, close the cursor and the connection as follows:

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



## Viewing Information About Imported Data

This example shows how to view information about imported data from the `dbtoolboxdemo` data source and close the connection to the database using the following Database Toolbox functions:

- `attr`
- `close`
- `cols`
- `columnnames`
- `rows`
- `width`

For more information on these functions, see `matlab\toolbox\database\dbdemos\dbinfodemo.m`.

### 1 Open the cursor and connection if needed:

```
conn = database('dbtoolboxdemo', '', '');  
curs = exec(conn, 'select productDescription from productTable');  
setdbprefs('DataReturnFormat','cellarray');  
curs = fetch(curs, 10);
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see `database`.

### 2 Use `rows` to return the number of rows in the data set:

```
numrows = rows(curs)  
numrows =  
    10
```

### 3 Use `cols` to return the number of columns in the data set:

```
numcols = cols(curs)  
numcols =  
    1
```

- 4 Use `colnames` to return the names of the columns in the data set:

```
colnames = colnames(curs)
colnames =
    'productDescription'
```

- 5 Use `width` to return the column width, or size of the field, for the specified column number:

```
colsize = width(curs, 1)
colsize =
    50
```

- 6 Use `attr` to view multiple attributes for a column:

```
attributes = attr(curs)

attributes =

    fieldName: 'productDescription'
    typeName: 'VARCHAR'
    typeValue: 12
columnWidth: 50
precision: []
    scale: []
currency: 'false'
readOnly: 'false'
nullable: 'true'
Message: []
```

---

**Tip** To import multiple columns, include a `colnum` argument in `attr` to specify the number of columns whose information you want.

---

- 7 Close the cursor.

```
close(curs);
```

- 8 Continue with the next example. To stop working now and resume working on the next example at a later time, close the connection.

```
close(conn);
```

# Exporting 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.

You learn to use the following Database Toolbox functions:

- get
- fastinsert
- setdbprefs

For more information on these functions, see `matlab\toolbox\database\dbdemos\dbinsertdemo.m`.

- 1 Connect to the data source, `dbtoolboxdemo`, if needed:

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see `database`.

- 2 Use `setdbprefs` to set the format for retrieved data to numeric:

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

- 3 Import ten rows of data the March column of data from the `salesVolume` table.

```
curs = exec(conn, 'select March from salesVolume');  
curs = fetch(curs);
```

- 4 Assign the data to the **MATLAB workspace variable** `AA`:

```
AA = curs.Data
AA =
```

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

- 5** Calculate the sum of the March sales and assign the result to the variable `sumA`:

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

```
14606
```

- 6** Assign the month and sum of sales to a cell array to export to a database. Put the month in the first cell of `exdata`:

```
exdata(1,1) = {'March'}
exdata =
    'March'
```

Put the sum in the second cell of `exdata`:

```
exdata(1,2) = {sumA}
exdata =
    'March'    [14606]
```

- 7** 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`:

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

- 8** Use the `get` function to determine the current 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 update; if it is `on`, data is automatically committed to the database.

```
get(conn, 'AutoCommit')
ans =
    on
```

The `AutoCommit` flag is set to `on`, so the exported data is automatically committed to the database.

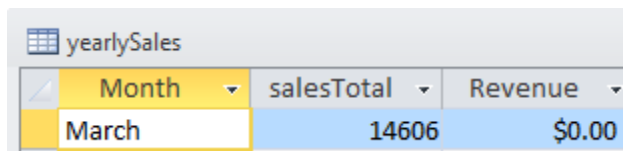
- 9** Use the `fastinsert` function to export the data into the `yearlySales` table. Pass the following arguments to this function:

- `conn`, the connection object for the database
- `yearlySales`, the name of the table to which you are exporting data
- The cell arrays `colnames` and `exdata`

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

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

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



Month	salesTotal	Revenue
March	14606	\$0.00

- 11** Close the cursor.

```
close(curs);
```

- 12** Continue with the next example (“Replacing Existing Database Data with Exported Data” on page 4-12). To stop now and resume working with the next example at a later time, close the connection.

```
close(conn);
```

## Replacing Existing Database Data with Exported Data

This example updates the Month field that you previously imported (“Exporting Data to New Record in Database” on page 4-8) into the `yearlySales` table of the `dbtoolboxdemo` data source using the following Database Toolbox functions:

- `close`
- `update`

For more information on these functions, see `matlab\toolbox\database\dbdemos\dbupdatedemo.m`.

- 1** Change the month in `yearlySales` table from March to March2010. Assign the new month value to the `newdata` cell array.

```
colnames = {'Month'};  
newdata = {'March2010'}  
newdata =  
    'March2010'
```

- 2** Specify the record to update in the database by defining a SQL `where` statement and assigning it to the variable `whereclause`. The record to update is the record whose Month is March. Because the date string is within a string, it is embedded within two single quotation marks rather than one.

```
whereclause = 'where Month = ''March'''  
whereclause =  
    where Month = 'March'
```

- 3** Export the data, replacing the record whose Month is March.

```
update(conn, 'yearlySales', colnames, newdata, whereclause)
```

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



yearlySales			
Month	salesTotal	Revenue	
March2010	14606	\$0.00	

- 5 Disconnect from the database.

```
close(conn);
```

# Exporting 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.

You use the following Database Toolbox functions:

- `fastinsert`
- `setdbprefs`

For more information on these functions, see `matlab\toolbox\database\dbdemos\dbinsert2demo.m`.

- 1 Ensure that the `dbtoolboxdemo` data source is writable, that is, not read only.
- 2 Use the `database` function to connect to the data source, assigning the returned connection object as `conn`. Pass the following arguments to this function:
  - `dbtoolboxdemo`, the name of the data source
  - `username` and `password`, which are passed as empty strings because no user name or password is required to access the database

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see `database`.

- 3 Use the `setdbprefs` function to specify preferences for the retrieved data. Set the data return format to `numeric` and 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 also be `numeric`.

**4** Import data from the `salesVolume` table.

```
curs = exec(conn, 'select * from salesVolume');  
curs = fetch(curs);
```

**5** Use `columnnames` to view the column names in the fetched data set:

```
columnnames(curs)  
ans =  
    'StockNumber'    'January'    'February'    'March'    'April',  
    'May'    'June'    'July'    'August'    'September'    'October',  
    'November'    'December'
```

**6** View the data for January (column 2).

```
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** Use `m` and `n` to compute monthly totals. 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
```

View the result.

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

- 9** Create a string array containing the column names into which you want to insert the data, and assign the array to the variable `colnames`.

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

- 10** Use `fastinsert` to insert the data into the `yearlySales` table:

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

- 11** To verify that the data was imported correctly, in Microsoft Access, view the `yearlySales` table from the tutorial database.

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

- 12** Close the cursor and the database connection.

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

## Exporting Data Using the Bulk Insert Command

### In this section...

“Bulk Insert to Oracle” on page 4-18

“Bulk Insert to Microsoft® SQL Server® 2005” on page 4-20

“Bulk Insert to MySQL” on page 4-22

### Bulk Insert to Oracle

This example demonstrates how to export data to the Oracle server using the bulk insert command. To follow 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 =

    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 to Microsoft SQL Server 2005

**1** Connect to the Microsoft SQL Server. For JDBC driver use, add the jar file to the MATLAB javaclasspath.

```
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
```

**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 =

    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 to MySQL

### 1 Connect to the MySQL server. For JDBC driver use, add the jar file to the MATLAB javaclasspath.

```
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 =

    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)
```

## Retrieving 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 more information, 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 data you imported.

```
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 more information on `parsebinary`, enter `help parsebinary` or view its file in the MATLAB Editor/Debugger by entering `open parsebinary`.

## Working with Database Metadata

### In this section...

“Accessing Metadata” on page 4-27

“Resultset Metadata Objects” on page 4-33

### Accessing Metadata

In this example, you use the following Database Toolbox functions to access metadata:

- `dmd`
- `get`
- `supports`
- `tables`

**1** Connect to the `dbtoolboxdemo` data source.

```
conn = database('dbtoolboxdemo', '', '')
conn =
    Instance: 'dbtoolboxdemo'
    UserName: ''
    Driver: []
    URL: []
    Constructor: [1x1 ...
    com.mathworks.toolbox.database.databaseConnect]
    Message: []
    Handle: [1x1 sun.jdbc.odbc.JdbcOdbcConnection]
    TimeOut: 0
    AutoCommit: 'on'
    Type: 'Database Object'
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see `database`.

**2** Use the `dmd` function to create a database metadata object `dbmeta` and return its handle, or identifier:

```
dbmeta = dmd(conn)
```

```
dbmeta = DMDHandle: ...  
[1x1 sun.jdbc.odbc.JdbcOdbcDatabaseMetaData]
```

- 3** Use the `get` function to assign database properties data, `dbmeta`, to the variable `v`:

```
v = get(dbmeta)  
v =  
    AllProceduresAreCallable: 1  
    AllTablesAreSelectable: 1  
    DataDefinitionCausesTransactionCommit: 1  
    DataDefinitionIgnoredInTransactions: 0  
    DoesMaxRowSizeIncludeBlobs: 0  
    Catalogs: {4x1 cell}  
    CatalogSeparator: '.'  
    CatalogTerm: 'DATABASE'  
    DatabaseProductName: 'ACCESS'  
    DatabaseProductVersion: '04.00.0000'  
    DefaultTransactionIsolation: 2  
    DriverMajorVersion: 2  
    DriverMinorVersion: 1  
    DriverName: [1x31 char]  
    DriverVersion: '2.0001 (04.00.6200)'  
    ExtraNameCharacters: [1x29 char]  
    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: [1x461 char]
    StringFunctions: [1x91 char]
StoresLowerCaseIdentifiers: 0
StoresLowerCaseQuotedIdentifiers: 0
StoresMixedCaseIdentifiers: 0
StoresMixedCaseQuotedIdentifiers: 1
StoresUpperCaseIdentifiers: 0
StoresUpperCaseQuotedIdentifiers: 0
    SystemFunctions: ''
    TableTypes: {13x1 cell}
    TimeDateFunctions: [1x111 char]
    TypeInfo: {16x1 cell}
    URL: ...
'jdbc:odbc:dbtoolboxdemo'
    UserName: 'admin'
NullPlusNonNullIsNull: 0
NullsAreSortedAtEnd: 0
NullsAreSortedAtStart: 0
NullsAreSortedHigh: 0
NullsAreSortedLow: 1
UsesLocalFilePerTable: 0
UsesLocalFiles: 1
```

---

**Tip** For more information about the database metadata properties returned by `get`, see the methods of the `DatabaseMetaData` object on the Oracle Java Web site:

[http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/DatabaseMetaData.ht](http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/DatabaseMetaData.html)

---

- 4** Some information is too long to fit in the display area of the field, so the size of the field data appears instead. The Catalogs element is shown as a 4-by-1 cell array. View the Catalog information.

```
v.Catalogs
```

```
ans =
'D:\Work\databasetoolboxfiles\tutorial'
'D:\Work\databasetoolboxfiles\tutorial_copy'
```

- 5** Use the supports function to see what properties this database supports:

```
a = supports(dbmeta)
a =
AlterTableWithAddColumn: 1
AlterTableWithDropColumn: 1
ANSI92EntryLevelSQL: 1
ANSI92FullSQL: 0
ANSI92IntermediateSQL: 0
CatalogsInDataManipulation: 1
CatalogsInIndexDefinitions: 1
CatalogsInPrivilegeDefinitions: 0
CatalogsInProcedureCalls: 0
CatalogsInTableDefinitions: 1
ColumnAliasing: 1
Convert: 1
CoreSQLGrammar: 0
CorrelatedSubqueries: 1
DataDefinitionAndDataManipulationTransactions: 1
DataManipulationTransactionsOnly: 0
DifferentTableCorrelationNames: 0
ExpressionsInOrderBy: 1
ExtendedSQLGrammar: 0
FullOuterJoins: 0
GroupBy: 1
GroupByBeyondSelect: 1
GroupByUnrelated: 0
IntegrityEnhancementFacility: 0
LikeEscapeClause: 0
LimitedOuterJoins: 0
```

```
MinimumSQLGrammar: 1
MixedCaseIdentifiers: 1
MixedCaseQuotedIdentifiers: 0
MultipleResultSets: 0
MultipleTransactions: 1
NonNullableColumns: 0
OpenCursorsAcrossCommit: 0
OpenCursorsAcrossRollback: 0
OpenStatementsAcrossCommit: 1
OpenStatementsAcrossRollback: 1
OrderByUnrelated: 0
OuterJoins: 1
PositionedDelete: 0
PositionedUpdate: 0
SchemasInDataManipulation: 0
SchemasInIndexDefinitions: 0
SchemasInPrivilegeDefinitions: 0
SchemasInProcedureCalls: 0
SchemasInTableDefinitions: 0
SelectForUpdate: 0
StoredProcedures: 1
SubqueriesInComparisons: 1
SubqueriesInExists: 1
SubqueriesInIns: 1
SubqueriesInQuantifieds: 1
TableCorrelationNames: 1
Transactions: 1
Union: 1
UnionAll: 1
```

A 1 for a given property indicates that the database supports that property; a 0 means that the database does not support the property.

---

**Tip** For more information about properties that the database supports, see the methods of the DatabaseMetaData object on the Oracle Java Web site at <http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/DatabaseMetaData.html>.

---

**6** Alternatively, use the `tables` function to retrieve metadata, such as the names and types of the tables in a catalog in the database. Pass the following arguments to this function:

- `dbmeta`, the name of the database metadata object.
- `tutorial`, the name of the catalog from which you want to retrieve table names.

```
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'
```

**7** Close the database connection.

```
close(conn)
```

## Resultset Metadata Objects

Use the `resultset` function to create resultset objects for cursor object. Then, use the `rsmd` function to get metadata information about the resultset objects.

For more information, see the `resultset` and `rsmd` function reference pages.

## Using Driver Functions

This example uses the following Database Toolbox functions to create driver and drivermanager objects, and to get and set their properties:

- `driver`
- `drivermanager`
- `get`
- `isdriver`
- `set`

---

**Note** There is no equivalent MATLAB example available because this example relies on a specific system-to-JDBC connection and database. Your configuration is different from the one in this example, so you cannot run these examples exactly as written. Instead, substitute appropriate values for your own system. See your database administrator for more information.

---

**1** Connect to the database.

```
c = database('orc1','scott','tiger',...  
            'oracle.jdbc.driver.OracleDriver',...  
            'jdbc:oracle:thin:@144.212.123.24:1822:');
```

**2** Use the `driver` function to construct a driver object and return its handle, for a specified database URL string of the form `jdbc:subprotocol:subname`.

```
d = driver('jdbc:oracle:thin:@144.212.123.24:1822:')  
DriverHandle: [1x1 oracle.jdbc.driver.OracleDriver]
```

**3** Use the `get` function to get information, such as version data, for the driver object.

```
v = get(d)  
v =  
MajorVersion: 1
```

```
MinorVersion: 0
```

- 4** Use `isdriver` to verify that `d` is a valid JDBC driver object.

```
isdriver(d)
ans =
    1
```

This result shows that `d` is a valid JDBC driver object. If it is a not valid JDBC driver object, the returned result is 0.

- 5** Use the `drivermanager` function to create a `drivermanager` object `dm`.

```
dm = drivermanager
```

- 6** Get properties of the `drivermanager` object.

```
v = get(dm)
v =
    Drivers: {'sun.jdbc.odbc.JdbcOdbcDriver@761630' ...
             [1x38 char]}
    LoginTimeout: 0
    LogStream: []
```

- 7** Set the `LoginTimeout` value to 10 for all drivers loaded during this session.

```
set(dm, 'LoginTimeout', 10)
```

Verify the `LoginTimeout` value.

```
v = get(dm)
v =
    Drivers: {'sun.jdbc.odbc.JdbcOdbcDriver@761630'}
    LoginTimeout: 10
    LogStream: []
```

# 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
- Driver
- Drivermanager
- 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.

## Using the exec Function

### In this section...

- “About the exec Function” on page 4-39
- “Using Cursor Objects” on page 4-39
- “Working with Microsoft® Excel®” on page 4-40
- “Database Considerations” on page 4-40

### About the exec Function

Use the `exec` function to execute an SQL statement and return the database cursor object. Here are some general points about using `exec`:

- Use Database Explorer to query databases as an alternative to using `exec`.
- `exec` supports the native ODBC interface.
- Unless noted in this reference page, the `exec` function supports all valid SQL statements, such as nested queries.
- The `sqlquery` argument can be a stored procedure for the database connection of the form `{call sp_name (parm1,parm2,...)}`.
- Use `exec` when the stored procedure returns one or more result sets. For procedures that return output parameters, use `runstoredprocedure`.

### Using Cursor Objects

- Check `curs.Message` to find any error messages returned from the database after query execution. If you would like the error messages to be thrown to the MATLAB command prompt, use `setdbprefs` as follows.

```
setdbprefs('Errorhandling','report');  
curs = exec(conn,'select * invalidtablename')
```

To store error messages in `curs.Message` instead of sending them to the MATLAB command prompt, use `setdbprefs` as follows.

```
setdbprefs('Errorhandling','store');
```

- After opening a cursor, use `fetch` to import data from the cursor. Use `resultset`, `rsmd`, and access the `Statement` property to get properties of the cursor.
- You can have multiple cursors open at one time.
- A cursor stays open until you close it using the `close` function.

### Working with Microsoft Excel

For Microsoft Excel, tables in `sqlquery` are Excel worksheets. By default, some worksheet names include `$`. To select data from a worksheet with this name format, use a SQL statement of the form `select * from "Sheet1$" (or 'Sheet1$')`.

### Database Considerations

- The order of records in your database is not constant. Use values in column names to identify records. Use the SQL `ORDER BY` command to sort records.
- 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 might experience issues with text field formats in the Microsoft SQL Server database management system. Workarounds for these issues are as follows:
  - 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#')
```

- 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#')
```

## Using the fetch Function

### In this section...

“About the fetch Function” on page 4-42

“fetch Workflow” on page 4-42

“Using fetch with a Cursor Object” on page 4-43

“Database Considerations” on page 4-44

### About the fetch Function

Use the `fetch` function to import data into the MATLAB workspace. Here are some general points about using `fetch`:

- Use Database Explorer to retrieve data as an alternative to using `fetch`.
- `fetch` supports the native ODBC interface.

### fetch Workflow

The `fetch` function runs the appropriate processes to retrieve data depending on what object you provide to it as an input argument. This function works with cursor objects and database connection objects for ODBC/JDBC bridge and JDBC interfaces. This function works with cursor objects only for the native ODBC interface.

For the JDBC database driver, use the `database` function to establish a database connection.

```
conn = database(...)
```

Then, `fetch` runs when you pass a cursor object, `curs`, to retrieve as an argument.

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

The `fetch` function runs when you pass a database object, `conn`, to retrieve as an argument.

```
fetch(conn,sqlquery)
```

---

**Note** You can pass `conn` as an input argument to `fetch` when you are using an ODBC/JDBC bridge or a JDBC interface. For the native ODBC interface, use `curs` as the input argument.

---

To create a database connection using the native ODBC interface, use `database.ODBCConnection`.

```
conn = database.ODBCConnection(...)
```

Then, the `fetch` function runs when you pass a native ODBC cursor object, `curs`, to retrieve as an argument.

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

When `fetch` returns a cursor object, you can run many other functions, such as `get` and `rows`. To import data into the MATLAB workspace without metadata, use `fetch` with a database connection object as the input argument.

## Using fetch with a Cursor Object

- `fetch` returns data stored in a MATLAB cell array, table, dataset, structure, or numeric matrix.
- When working with a JDBC or JDBC/ODBC bridge connection established using `database`, running `fetch` on the cursor object returns a new object of type `cursor` which points to the same underlying Java objects as the input cursor. It is therefore best practice to overwrite the input cursor object. This practice results in only one open cursor object, which consumes less memory than multiple open cursor objects.

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

After this, you simply need to close this one object. Creating a different variable for the output cursor object will unnecessarily create 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 or not 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)
```

- The next time `fetch` is run, records are imported starting with the row following the specified number of rows in `rowlimit`. If you specify a row limit of 0, all the rows of data are fetched.
- Fetching large amounts of data can result in memory or speed issues. Use `rowlimit` to limit how much data you retrieve at once.
- If 'FetchInBatches' is set to 'yes' in the preferences using `setdbprefs`, `fetch` incrementally fetches the number of rows specified in the 'FetchBatchSize' setting until all the rows returned by the query are fetched, or until the limited number of rows are fetched, if `rowlimit` is specified. Use this method when fetching a large number of rows from the database.

---

**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`.

---

### Database Considerations

- The order of records in your database does not remain constant. Use the `SQL ORDER BY` command in your `sqlquery` statement to sort data.



# Functions — Alphabetical List

---

# attr

---

**Purpose** Retrieve attributes of columns in fetched data set

**Syntax**  
`attributes = attr(curs, colnum)`  
`attributes = attr(curs)`

**Description** `attributes = attr(curs, colnum)` retrieves attribute information for the column number `colnum` in the fetched data set `curs`.

`attributes = attr(curs)` retrieves attribute information for all columns in the fetched data set `curs` and stores the data in a cell array.

`attributes = attr(colnum)` displays attributes of column `colnum`.

A list of returned attributes appears in the following table.

Attribute	Description
<code>fieldName</code>	Name of the column.
<code>typeName</code>	Data type.
<code>typeValue</code>	Numerical representation of the data type.
<code>columnWidth</code>	Size of the field.
<code>precision</code>	Precision value for floating and double data types; an empty value is returned for strings.
<code>scale</code>	Precision value for real and numeric data types; an empty value is returned for strings.
<code>currency</code>	If <code>true</code> , data format is currency.
<code>readOnly</code>	If <code>true</code> , data cannot be overwritten.
<code>nullable</code>	If <code>true</code> , data can be NULL.
<code>Message</code>	Error message returned by <code>fetch</code> .

## Examples **Example 1 – Get Attributes for One Column**

Get column attributes for the fourth column of a fetched data set:

```
attr(curs, 4)
```

```
ans =
  fieldName: 'Age'
  typeName: 'LONG'
  typeValue: 4
  columnWidth: 11
  precision: []
  scale: []
  currency: 'false'
  readOnly: 'false'
  nullable: 'true'
  Message: []
```

## Example 2 – Get Attributes for All Columns

- 1 Get column attributes for curs and assign them to attributes.

```
attributes = attr(curs)
```

- 2 View the attributes of column 4.

```
attributes(4)
ans =
  fieldName: 'Age'
  typeName: 'LONG'
  typeValue: 4
  columnWidth: 11
  precision: []
  scale: []
  currency: 'false'
  readOnly: 'false'
  nullable: 'true'
  Message: []
```

### See Also

`cols` | `columnnames` | `columns` | `fetch` | `dmd` | `get` | `tables` | `width`

# bestrowid

---

**Purpose** Unique identifier for row in database table

**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`

**Purpose** Clear warnings for database connection or resultset

**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`

# close

---

**Purpose** Close database connection, cursor, or resultset object

**Syntax** `close(object)`

**Description** `close(object)` closes `object`, which frees up resources. Allowable objects for `close` are listed in the following table.

<b>Object</b>	<b>Description</b>	<b>Action Performed by close (object)</b>
<code>conn</code>	Database connection object or native ODBC database connection object	Closes <code>conn</code>
<code>curs</code>	Cursor object or native ODBC cursor object	Closes <code>curs</code>
<code>rset</code>	Resultset object	Closes <code>rset</code>

Database connections, cursors, and resultset objects remain open until you close them using the `close` function. Always close a cursor, connection, or resultset when you finish using it. Close a cursor before closing the connection used for that cursor.

---

**Note** The MATLAB session closes open cursors and connections when exiting, but the database might not free up the cursors and connections.

---

---

**Tip** For command-line help on `close`, use the overloaded methods:

```
help database/close  
help cursor/close  
help resultset/close
```

---

## Examples

Close the cursor `curs` and the connection `conn`.

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

## See Also

`fetch` | `database` | `exec` | `resultset`

# cols

---

**Purpose** 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 three columns in the fetched data set `curs`.

```
numcols = cols(curs)
```

```
numcols =  
    3
```

**See Also** `attr` | `columnnames` | `columnprivileges` | `columns` | `fetch` | `get`  
| `rows` | `width`

**How To** • “Using the Native ODBC Database Connection” on page 2-12



**Purpose**

Retrieve names of columns in fetched data set

**Syntax**

```
FIELDSTRING = columnnames(CURSOR)
FIELDSTRING = columnnames(CURSOR, BCELLARRAY)
```

**Description**

FIELDSTRING = columnnames(CURSOR) returns the column names of the data selected from a database table. The column names are enclosed in quotes and separated by commas. (The columnnames function is not supported for a cursor object returned by the fetchmulti function.)

FIELDSTRING = columnnames(CURSOR, BCELLARRAY) returns the column names as a cell array of strings when BCELLARRAY is set to true.

**Examples**

- 1 Run a SQL query to return all columns from the Microsoft Access dbtoolboxdemo data source database suppliers table:

```
sql = 'select * from suppliers'
cursor = exec(connection, sql)
cursor = fetch(cursor)
```

- 2 Use columnnames to retrieve all column names for the selected columns:

```
fieldString = columnnames(cursor)
fieldString =
'SupplierNumber', 'SupplierName', 'City', 'Country', 'FaxNumber'
```

**See Also**

attr | cols | columnprivileges | columns | fetch | get | width

# columnprivileges

---

**Purpose** 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` | `columns` | `columnnames` | `dmd` | `get`

**Purpose**

Return database table column names

**Syntax**

```
l = columns(dbmeta, 'cata')
l = columns(dbmeta, 'cata', 'sch')
l = columns(dbmeta, 'cata', 'sch', 'tab')
```

**Description**

`l = columns(dbmeta, 'cata')` returns a list of all column names in the catalog `cata` for the database whose database metadata object is `dbmeta`.

`l = columns(dbmeta, 'cata', 'sch')` returns a list of all column names in the schema `sch`.

`l = columns(dbmeta, 'cata', 'sch', 'tab')` returns a list of columns for the table `tab`.

**Examples**

**1** Run `columns` for the arguments shown:

```
l = columns(dbmeta, 'orcl', 'SCOTT')
l =
    'BONUS'          {1x4 cell}
    'DEPT'           {1x3 cell}
    'EMP'            {1x8 cell}
    'SALGRADE'       {1x3 cell}
    'TRIAL'          {1x3 cell}
```

The results show the names of the five tables in `dbmeta`, and cell arrays containing the column names in each table.

**2** View the column names for the `BONUS` table:

```
l{1,2}
ans =
    'ENAME'    'JOB'    'SAL'    'COMM'
```

**See Also**

`attr` | `bestrowid` | `cols` | `columnnames` | `columnprivileges` | `dmd`  
| `get` | `versioncolumns`

# commit

---

**Purpose** 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`.

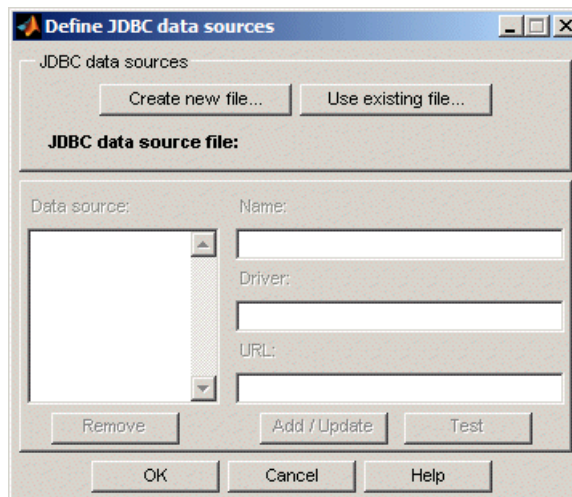
```
fastinsert(conn, 'DEPT', {'DEPTNO'; 'DNAME'; 'LOC'}, ...
exdata)
```

**2** Commit this data.

```
commit(conn)
```

**See Also** `database` | `exec` | `fastinsert` | `get` | `rollback` | `update`

- Purpose** 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 “Configure JDBC Data Sources”.

---

**Tip** Use the database function to define JDBC data sources programmatically.

---

**See Also** database | querybuilder

# crossreference

---

## Purpose

Retrieve information about primary and foreign keys

## 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

Column	Description	Value
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/1.4.2/docs/api/java/sql/DatabaseMetaData>

---

## crossreference

---

### **See Also**

dmd | exportedkeys | get | importedkeys | primarykeys



<b>Purpose</b>	Import data into MATLAB Workspace from cursor object created by <code>exec</code>
<b>Alternatives</b>	Retrieve data using Database Explorer ( <code>dexplore</code> ).
<b>Syntax</b>	<pre>curs = fetch(curs,rowLimit) curs = fetch(curs)</pre>
<b>Description</b>	<p><code>curs = fetch(curs,rowLimit)</code> imports rows of data into the object <code>curs</code> from the open SQL cursor <code>curs</code>, up to the maximum <code>rowLimit</code>.</p> <p><code>curs = fetch(curs)</code> imports rows of data from the open SQL cursor <code>curs</code> into the object <code>curs</code>, up to <code>rowLimit</code>. Use the <code>set</code> function to specify <code>rowLimit</code>.</p> <p>Data is stored in a MATLAB cell array, table, dataset, structure, or numeric matrix. It is a best practice to assign the object returned by <code>fetch</code> to the variable <code>curs</code> from the open SQL cursor. This practice results in only one open cursor object, which consumes less memory than multiple open cursor objects.</p> <p>The next time <code>fetch</code> is run, records are imported starting with the row following the specified <code>rowLimit</code>. If you specify a <code>rowLimit</code> of 0, all the rows in the resultset are fetched.</p> <p>If 'FetchInBatches' is set to 'yes' in the preferences using <code>setdbprefs</code>, <code>cursor.fetch</code> incrementally fetches the number of rows specified in the 'FetchBatchSize' setting until all the rows returned by the query are fetched, or until <code>rowLimit</code> number of rows are fetched, if <code>rowLimit</code> is specified. Use this method when fetching a large number of rows from the database.</p> <p>Fetching large amounts of data can result in memory or speed issues. In this case, use <code>rowLimit</code> to limit how much data you retrieve at once.</p>

---

**Caution:** Leaving cursor and connection objects open or overwriting open objects can result in unexpected behavior. Once you are finished working with these objects, you must close them using `close`.

---

## Tips

- This page documents `fetch` for a cursor object. For more information about using `fetch`, `cursor.fetch`, and `database.fetch`, see `fetch`. Unless otherwise noted, `fetch` in this documentation refers to `cursor.fetch`, rather than `database.fetch`.
- `cursor.fetch` now supports the native ODBC interface.

## Examples

### Import All Rows of Data Using the Native ODBC Interface

Create a connection `conn` using the native ODBC interface and the `dbtoolboxdemo` data source.

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

```
conn =
```

```
ODBCConnection 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.

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

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

```
curs =
```

```
ODBCCursor 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'
```

Close the cursor object.

```
close(curs);
```

## Import All Rows of Data

Working with the `dbtoolboxdemo` data source, use `exec` to select data in column `City`, for example, in table `suppliers`. Then, use `fetch` to import all data from the SQL statement into the cursor object `curs`, and store the data in a cell array contained in the cursor object field `curs.Data`.

```
curs = exec(conn,'select City from suppliers');  
curs = fetch(curs)
```

```
curs =
```

```
Attributes: []  
Data: {10x1 cell}  
DatabaseObject: [1x1 database]  
RowLimit: 0  
SQLQuery: 'select City from suppliers'  
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 =
```

```
'New York'  
'London'  
'Adelaide'  
'Dublin'  
'Boston'  
'New York'  
'Wellesley'
```

```
'Nashua'
'London'
'Belfast'
```

Close the cursor object.

```
close(curs);
```

### Import a Specified Number of Rows

Working with the dbtoolboxdemo data source, use the `rowLimit` argument to retrieve only the first three rows of data.

```
curs = exec(conn,'select productdescription from producttable');
curs = fetch(curs,3)
```

```
curs =
```

```
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'
```

Close the cursor object.

```
close(curs);
```

### **Import Rows Iteratively Until You Retrieve All Data**

Working with the dbtoolboxdemo data source, use the rowLimit argument to retrieve the first two rows of data, and then rerun the import using a while loop, retrieving two rows at a time. Continue until you have retrieved all data, which occurs when curs.Data is 'No Data'.

```
curs = exec(conn,'select productdescription from producttable');  
% 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'
```

Close the cursor object.

```
close(curs);
```

## Import Numeric Data

Working with the dbtoolboxdemo data source, import a column of numeric data, using the `setdbprefs` function to specify numeric as the format for the retrieved data.

```
curs = exec(conn,'select unitCost from productTable');
setdbprefs('DataReturnFormat','numeric')
curs = fetch(curs,3);
curs.Data
```

```
ans =

    13
     5
    16
```

Close the cursor object.

```
close(curs);
```

## Import BOOLEAN Data

Import data that includes a `BOOLEAN` field, using the `setdbprefs` function to specify `cellarray` as the format for the retrieved data.

```
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
```

Close the cursor object.

```
close(curs);
```

### Perform Incremental Fetch

Working with the dbtoolboxdemo data source, retrieve data incrementally to avoid Java heap errors. Use `cursor.fetch` with the `setdbprefs` properties for `FetchInBatches` and `FetchBatchSize` to fetch large data sets.

```
setdbprefs('FetchInBatches','yes');
setdbprefs('FetchBatchSize','2');
conn = database('dbtoolboxdemo','','');
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'
```

`cursor.fetch` internally retrieves data in increments of two rows at a time. Tune the `FetchBatchSize` setting depending on the size of the result set 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. The optimal value for `FetchBatchSize` is decided based on several factors like the size per row being retrieved, the Java heap memory value, the driver's default fetch size, and system architecture, and hence, can vary from site to site.

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.`

You can exercise 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, `cursor.fetch` retrieves the first three rows of `productTable`, two rows at a time.

Close the cursor object.

```
close(curs);
```

**See Also**

`attr` | `cols` | `columnnames` | `database` | `database.fetch` | `exec` | `fetch` | `fetchmulti` | `get` | `logical` | `rows` | `resultset` | `set` | `width`

**Tutorials**

- “Getting Started with Visual Query Builder” on page 3-2
- “Preference Settings for Large Data Import” on page 3-10

**How To**

- “Working with Visual Query Builder”
- “Retrieving BINARY and OTHER Data Types” on page 3-51
- “Using the Native ODBC Database Connection” on page 2-12

# database

---

## Purpose

Connect to database

## Syntax

```
conn = database(instance,username,password)
conn = database.ODBCConnection(instance,username,password)

conn = database(instance,username,password,driver,
                databaseurl)

conn = database(instance,username,password,Name,Value)
```

## Description

`conn = database(instance,username,password)` returns a database connection object for the connection to the ODBC data source setup, `instance`, via an ODBC driver.

`conn = database.ODBCConnection(instance,username,password)` returns a database connection object for the connection to the ODBC data source setup, `instance`, via a native ODBC interface.

`conn = database(instance,username,password,driver,databaseurl)` connects to the database, `instance`, via a JDBC driver.

`conn = database(instance,username,password,Name,Value)` connects to the database, `instance`, via JDBC driver with connection properties specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **instance - Data source setup or database name**

string

Data source setup or database name, specified as a string. Specify a data source for ODBC connection, and the database name for JDBC connection.

### **username - User name**

string

User name required to access the database, specified as a string. If no user name is required, specify empty strings, ''.

### **password - Password**

string

Password required to access the database, specified as a string. If no password is required, specify empty strings, ''.

### **driver - JDBC driver name**

string

JDBC driver name, specified as a string. This is the name of the Java driver that implements the `java.sql.Driver` interface. It is part of the JDBC driver name and database connection URL.

### **databaseurl - Database connection URL**

string

Database connection URL, specified as a string. This is a vendor-specific URL that is typically constructed using connection properties like server name, port number, database name, and so on. It is part of the 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 the following strings:

- '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) | string

Database server name or address, specified as the comma-separated pair consisting of 'Server' and a string value.

**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 the following strings:

- '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 the following strings:

- 'thin'
- 'oci'

**Example:** 'DriverType', 'thin'

### 'URL' - Connection URL

string

Connection URL, specified as the comma-separated pair consisting of 'URL' and a string value. If you specify URL, you might not need to specify any other properties.

## Output Arguments

### conn - Database connection

Database connection object

Database connection, returned as a database connection object. The database connection object has the following properties:

- Instance
- UserName
- Driver
- URL
- Constructor
- Message
- Handle

- TimeOut
- AutoCommit
- Type

The native ODBC database connection object, `database.ODBCConnection`, excludes `Driver`, `URL`, and `Constructor` properties. For `database.ODBCConnection`, the `Type` property is equal to `database.ODBCConnection` object. The `Handle` property for a `database.ODBCConnection` object is `database.internal.ODBCConnectHandle`, and for JDBC/ODBC bridge connection, it is `sun.jdbc.odbc.JdbcOdbcConnection`. For ODBC, the `Instance` property contains the data source name, and, for JDBC, the `Instance` property contains the database name.

## Tips

- Use `logintimeout` before `database` to set the maximum time for a connection attempt.
- Alternatively use Visual Query Builder to connect to databases.
- 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).

## Examples

### Connect Using the Native ODBC Interface

Connect to the `dbtoolboxdemo` 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')
```



```
conn =
```

```
ODBCConnection with properties:
```

```

    Instance: 'dbtoolboxdemo'
    UserName: 'admin'
    Message: []
    Handle: [1x1 database.internal.ODBCConnectHandle]
    TimeOut: 0
    AutoCommit: 0
    Type: 'ODBCConnection Object'
```

database.ODBCConnection returns conn as database.ODBCConnection object.

Close the database connection, conn.

```
close(conn);
```

## ODBC Connection

Connect to the dbtoolboxdemo database.

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

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

```
conn =
```

```

    Instance: 'dbtoolboxdemo'
    UserName: 'admin'
    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.

Close the database connection, conn.

```
close(conn);
```

## **Oracle JDBC Connection Using URL**

Connect to an Oracle database using the JDBC driver.

Connect to the database, test\_db, using the user name, scott, and password, tiger. Use the JDBC driver, oracle.jdbc.driver.OracleDriver, to make the connection. The URL defined by the driver vendor is jdbc:oracle:oci7:.

```
conn = database('test_db','scott','tiger',...  
               'oracle.jdbc.driver.OracleDriver','jdbc:oracle:oci7:')
```

## **Oracle JDBC Connection Using Name-Value Connection Properties**

Connect to an Oracle database using the JDBC driver. Specify the vendor and connection options using name-value pair arguments.

Connect to the database, test\_db, using the user name, scott, and password, tiger. The Database Server machine name is remotehost and the port number that the server is listening on is 1234.

```
conn = database('test_db','scott','tiger','Vendor','Oracle',...  
               'DriverType','oci','Server','remotehost','PortNumber',1234)
```

## **MySQL JDBC Connection on the Default Port**

Connect to a MySQL database via a JDBC driver. Specify the vendor and connection options using name-value pair arguments.

Connect to the database, test\_db, on the machine, remotehost. Use the user name, root, and password, matlab.

```
conn = database('test_db','root','matlab','Vendor','MySQL',...
```

```
'Server', 'remotehost')
```

## **Microsoft Access Connection to a Database with .accdb Format**

Connect to a Microsoft Access database with .accdb format using an ODBC driver.

Connect to the database, MyDatabase.accdb, using dpath and url.

```
dbpath = ['C:\Data\Matlab\MyDatabase.accdb'];
url = [['jdbc:odbc:Driver={Microsoft Access Driver (*.mdb, *.accdb)};DSN='';DBQ='] dbpath];
con = database('','','','sun.jdbc.odbc.JdbcOdbcDriver', url);
```

## **PostgreSQL JDBC Connection to localhost on the Default Port**

Connect to a local PostgreSQL database using the JDBC driver. Specify the vendor and connection options using name-value pair arguments.

Connect to the database, test\_db, using the user name, postgres, and password, matlab.

```
conn = database('test_db', 'postgres', 'matlab', 'Vendor', 'PostgreSQL')
```

## **Microsoft SQL Server Windows Authenticated Database Connection**

Connect to a Microsoft SQL Server database with integrated Windows Authentication using a JDBC driver.

Close MATLAB if it is running.

Insert the path to the database driver JAR file in the javaclasspath.txt file. The updated path entry should include the full path to the driver. For example:

```
C:\DB_Drivers\sqljdbc_2.0\enu\sqljdbc4.jar
```

Insert the path to the folder containing sqljdbc\_auth.dll in the librarypath.txt file. The librarypath.txt file is located at:

```
$MATLABROOT\toolbox\local\librarypath.txt
```

The path entry should not include the file name `sqljdbc_auth.dll`:

```
C:\DB_Drivers\sqljdbc_2.0\enu\auth\x64
```

The `sqljdbc_auth.dll` file is installed in the following location:

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

where *<installation>* is the installation folder of the SQL server driver.

- If you are running a 32-bit Java Virtual Machine (JVM), then use the `sqljdbc_auth.dll` file in the x86 folder, even if the operating system is the x64 version.
- If you are running a 64-bit JVM on a x64 processor, then use the `sqljdbc_auth.dll` file in the x64 folder.
- If you are running a 64-bit JVM on a IA-64 processor, then use the `sqljdbc_auth.dll` file in the IA64 folder.

Start MATLAB.

Use the `AuthType` parameter to establish a Windows Authentication connection.

```
conn = database('dbName','','...', ...  
    'Vendor','Microsoft SQL Server','Server','servername',...  
    'AuthType','Windows')
```

## Definitions

### 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	<b>JDBC Driver:</b> com.informix.jdbc.IfxDriver <b>Database URL:</b> jdbc:informix-sqli://161.144.202.206:3000:INFORMIXSERVER=stars
Microsoft SQL Server 2005	<b>JDBC Driver:</b> com.microsoft.sqlserver.jdbc.SQLServerDriver <b>Database URL:</b> jdbc:sqlserver://localhost:port;database=databasename
MySQL	<b>JDBC Driver:</b> twz1.jdbc.mysql.jdbcMySqlDriver <b>Database URL:</b> jdbc:z1MySQL://natasha:3306/metrics <b>JDBC Driver:</b> com.mysql.jdbc.Driver <b>Database URL:</b> jdbc:mysql://devmetrics.mrkps.com/testing  To insert or select characters with encodings that are not default, append the string useUnicode=true&characterEncoding=... to the URL, where ... is any valid MySQL character encoding. For example, useUnicode=true&characterEncoding=utf8.
Oracle oci7 drivers	<b>JDBC Driver:</b> oracle.jdbc.driver.OracleDriver <b>Database URL:</b> jdbc:oracle:oci7:@rex
Oracle oci8 Drivers	<b>JDBC Driver:</b> oracle.jdbc.driver.OracleDriver <b>Database URL:</b> jdbc:oracle:oci8:@111.222.333.44:1521: <b>Database URL:</b> jdbc:oracle:oci8:@frug
Oracle 10 Connections with JDBC (Thin Drivers)	<b>JDBC Driver:</b> oracle.jdbc.driver.OracleDriver <b>Database URL:</b> jdbc:oracle:thin:
Oracle Thin Drivers	<b>JDBC Driver:</b> oracle.jdbc.driver.OracleDriver <b>Database URL:</b> jdbc:oracle:thin:@144.212.123.24:1822:

# database

---

Database	JDBC Driver Name and Database URL Example Syntax
PostgreSQL	<b>JDBC Driver:</b> org.postgresql.Driver <b>Database URL:</b> jdbc:postgresql://host:port/database
PostgreSQL with SSL Connection	<b>JDBC Driver:</b> org.postgresql.Driver <b>Database URL:</b> jdbc:postgresql:servername:dbname:ssl=true&sslfactory=org.postgresql.ssl.NonValidatingFactory& <i>The trailing &amp; is required.</i>
Sybase SQL Server and Sybase SQL Anywhere	<b>JDBC Driver:</b> com.sybase.jdbc.SybDriver <b>Database URL:</b> jdbc:sybase:Tds:yourhostname:yourportnumber/

## See Also

[close](#) | [dexplore](#) | [dmd](#) | [exec](#) | [fastinsert](#) | [get](#) | [getdatasources](#) | [isconnection](#) | [isreadonly](#) | [logintimeout](#) | [ping](#) | [querybuilder](#) | [supports](#) | [update](#)

## Concepts

- “Database Connection Error Messages” on page 2-8
- “Bringing Java Classes into MATLAB Workspace”
- “Using the Native ODBC Database Connection” on page 2-12

**Purpose** Get database catalog names

**Syntax** P = catalogs(conn)

**Description** P = catalogs(conn) returns the catalogs for the database connection conn.

**See Also** get | database.columns | database.schemas | database.tables

# database.columns

---

**Purpose** Get database table column names

**Syntax**

```
P = columns(conn)
P = columns(conn,C)
P = columns(conn,C,S)
P = columns(conn,C,S,T)
```

**Description**

P = columns(conn) returns all columns for all tables given the database connection conn.

P = columns(conn,C) returns all columns for all tables of all schemas for the given catalog C.

P = columns(conn,C,S) returns the columns for all tables for the given catalog C and schema S.

P = columns(conn,C,S,T) returns the columns for the given database connection conn, the catalog C, the schema S, and the table T.

**See Also** `get | database.schemas | database.tables`



<b>Purpose</b>	Execute SQL statement to import data into MATLAB workspace
<b>Syntax</b>	<pre>results = fetch(conn,sqlquery) results = fetch(conn,sqlquery,fetchbatchsize)</pre>
<b>Description</b>	<p><code>results = fetch(conn,sqlquery)</code> executes the SQL statement <code>sqlquery</code>, imports data for the open connection object <code>conn</code>, and returns the data to <code>results</code>. (For more information on SQL statements, see <code>exec</code>.)</p> <p><code>results = fetch(conn,sqlquery,fetchbatchsize)</code> imports <code>fetchbatchsize</code> rows of data at a time.</p>
<b>Input Arguments</b>	<p><b>conn</b> A database connection object.</p> <p><b>sqlquery</b> An SQL statement.</p> <p><b>fetchbatchsize</b> Specifies the number of rows of data to import at a time. Use <code>fetchbatchsize</code> when importing large amounts of data. Retrieving data in increments, as specified by <code>fetchbatchsize</code>, helps reduce overall retrieval time. If <code>fetchbatchsize</code> is not provided, a default value of <code>FetchBatchSize</code> is used. <code>FetchBatchSize</code> is set using <code>setdbprefs</code>.</p>
<b>Output Arguments</b>	<p><b>results</b> A cell array, table, dataset, structure, or numeric matrix depending on specifications set by <code>setdbprefs</code>.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• You call the <code>database.fetch</code> function with <code>fetch</code> rather than <code>database.fetch</code>. You implicitly call <code>database.fetch</code> by passing a database object, <code>conn</code>, to <code>fetch</code>. The <code>fetch</code> function also works with a cursor object. See <code>cursor.fetch</code>.</li></ul>

# database.fetch

---

- The order of records in your database does not remain constant. Use the SQL ORDER BY command in your sqlquery statement to sort data.

## Examples

### Import Data

Import the productDescription column from the productTable table in the dbtoolboxdemo database.

```
conn = database('dbtoolboxdemo','','');
setdbprefs('DataReturnFormat','cellarray')
results = fetch(conn,'select productdescription from producttable')
```

```
results =

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

If you experience speed and memory issues, use the fetchbatchsize argument.

View the size of the cell array into which the results were returned.

```
size(results)
```

```
ans =

    10     1
```

## Import Two Columns of Data and View Information About the Data

Import the InvoiceNumber and Paid columns from the Invoice table in the dbtoolboxdemo database.

```
conn = database('dbtoolboxdemo','','');
setdbprefs('DataReturnFormat','cellarray')
results = fetch(conn,['select InvoiceNumber, '...
'Paid from Invoice']);
```

View the size of the cell array into which the results were returned.

```
size(results)
```

```
ans =
```

```
    12     2
```

View the results for the first row of data.

```
results(1,:)
```

```
ans =
```

```
    [2101]    [0]
```

View the data type of the second element in the first row of data.

```
class(results{1,2})
```

```
ans =
```

```
logical
```

### See Also

`cursor.fetch` | `database` | `exec` | `fetch` | `logical`

### How To

- “Retrieving Image Data Types” on page 4-25

- “Preference Settings for Large Data Import” on page 3-10

**Purpose** Get database schema names

**Syntax** P = schemas(conn)

**Description** P = schemas(conn) returns the schema names for the database connection conn.

**See Also** get | database.catalogs | database.columns | database.tables

# database.tables

---

**Purpose** Get database table names

**Syntax**  
T = tables(conn)  
T = tables(conn,C)  
T = tables(conn,C,S)

**Description** T = tables(conn) returns all tables and table types for the database connection object conn.  
T = tables(conn,C) returns all tables and table types for all schemas of the given catalog name C.  
T = tables(conn,C,S) returns the list of tables and table types for the database with the catalog name C and schema name S.

**See Also** get | database.catalogs | database.schemas

<b>Purpose</b>	Export MATLAB data into database table
<b>Syntax</b>	<code>datainsert(connect,tablename,fieldnames,data)</code>
<b>Description</b>	<code>datainsert(connect,tablename,fieldnames,data)</code> inserts data from the MATLAB workspace into a database table.
<b>Tips</b>	You can also use the <code>fastinsert</code> function to export MATLAB data into a database table. The <code>fastinsert</code> function allows more flexibility in terms of the date and time string format of input data, but it is slower than <code>datainsert</code> .
<b>Input Arguments</b>	<p><b>connect</b> Database connection object.</p> <p><b>tablename</b> Database table.</p> <p><b>fieldnames</b> String array of database column names.</p> <p><b>data</b> MATLAB cell array or numeric matrix.</p> <p>If <code>data</code> is a cell array containing MATLAB dates, times, or timestamps, the dates must be date strings of the form <code>yyyy-mm-dd</code>, times must be time strings of the form <code>HH:MM:SS</code>, and timestamps must be strings of the form <code>yyyy-mm-dd HH:MM:SS.FFF</code>. <code>null</code> entries must be empty strings and any NaNs in the cell array must be converted to empty strings before calling <code>datainsert</code>.</p> <p>MATLAB date numbers and NaNs are supported for insert when <code>data</code> is a numeric array. Date numbers inserted into database date and time columns convert to <code>java.sql.Date</code>.</p>

# datainsert

---

## Examples

Export MATLAB cell array data into the field names col1, col2, and col2 in the 'inserttable' database table:

```
datainsert(connect,'inserttable',{'col1','col2','col2'},...  
          {33.5 8.77 '2010-07-04'})
```

---

Export data from a numeric matrix into a database table:

```
datainsert(connect,'inserttable',{'col1','col2','col2'},...  
          [33.5 8.77 734323])
```

## See Also

[fastinsert](#) | [insert](#) | [update](#)



**Purpose** Start SQL Database Explorer to import data

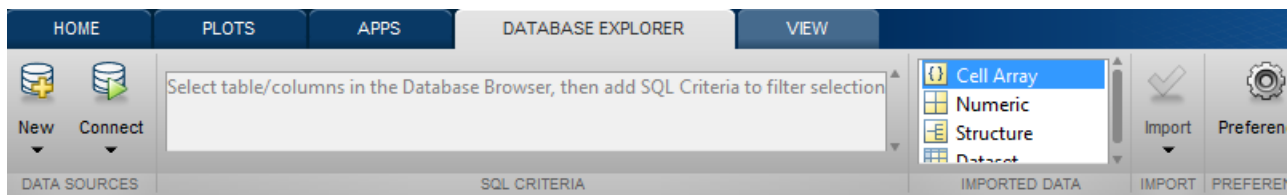
**Syntax** dexplore

**Description** dexplore starts Database Explorer, which is the Database Toolbox app for connecting to a database and importing data to the MATLAB workspace.

Database Explorer is an interactive app that allows you to:

- 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

**Examples** For more information on Database Explorer, after starting Database Explorer, click **Help** on the Database Explorer Toolstrip:



## Related Examples

- “Using Database Explorer” on page 3-61

# dmd

---

**Purpose** 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`

<b>Purpose</b>	Construct database driver object
<b>Syntax</b>	<code>d = driver('s')</code>
<b>Description</b>	<code>d = driver('s')</code> constructs a database driver object <code>d</code> from <code>s</code> , where <code>s</code> is a database URL string of the form <code>jdbc:odbc:name</code> or <code>name</code> . The driver object <code>d</code> is the first driver that recognizes <code>s</code> .
<b>Examples</b>	<code>d = driver('jdbc:odbc:thin:@144.212.123.24:1822:')</code> creates driver object <code>d</code> .
<b>See Also</b>	<code>get</code>   <code>isdriver</code>   <code>isjdbc</code>   <code>isurl</code>   <code>register</code>

# drivermanager

---

**Purpose** Construct database drivermanager object

**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` | `register` | `set`

---

<b>Purpose</b>	Execute SQL statement and open cursor
<b>Syntax</b>	<pre>curs = exec(conn,sqlquery) curs = exec(conn,sqlquery,qTimeout)</pre>
<b>Description</b>	<p><code>curs = exec(conn,sqlquery)</code> executes the SQL statement <code>sqlquery</code> for the database connection <code>conn</code> and returns the cursor object <code>curs</code>.</p> <p><code>curs = exec(conn,sqlquery,qTimeout)</code> executes the SQL statement with a timeout value <code>qTimeout</code>.</p>
<b>Input Arguments</b>	<p><b>conn - Database connection</b> connection object</p> <p>Database connection, specified as a database connection object created using database.</p> <p><b>sqlquery - SQL statement</b> SQL string</p> <p>SQL statement, specified as an SQL string to execute.</p> <p><b>Data Types</b> char</p> <p><b>qTimeout - Timeout value</b> scalar</p> <p>Timeout value, specified as a scalar denoting the maximum amount of time in seconds <code>exec</code> tries to execute the SQL statement, <code>sqlquery</code>.</p> <p><b>Data Types</b> double</p>
<b>Output Arguments</b>	<p><b>curs - Database cursor</b> database cursor object</p> <p>Database cursor, returned as a database cursor object. The properties of this object are different based on the database connection object.</p>

For an ODBC/JDBC bridge or a JDBC driver database connection, the cursor object has the following properties.

<b>Property</b>	<b>Description</b>
Attributes	Not used.
Data	Contains the resulting data after executing fetch.
DatabaseObject	Database connection object or <code>database.ODBCConnection</code> object that opened the cursor object.
RowLimit	Number of rows to fetch at a time.
SQLQuery	SQL statement to execute.
Message	Contains the error messages generated from executing the SQL statement. If this property is empty, then the SQL statement executed successfully.
Type	Database cursor object or <code>database.ODBCCursor</code> object type.
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.

For a native ODBC connection, the cursor object has only these properties from the previous list: `Data`, `RowLimit`, `SQLQuery`, `Message`, `Type`, and `Statement`.

## **Examples**

### **Select Data from a Database Table Using the Native ODBC Interface**

Create a connection `conn` using the native ODBC interface and the `dbtoolboxdemo` data source.

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

```
conn =
```

```
ODBCConnection 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 data from 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)
```

```
curs =
```

```
ODBCCursor 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 database.ODBCCursor object instead of a Database Cursor Object.

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

```
close(curs);
```

### **Select Data from a Database Table**

Using the dbtoolboxdemo data source, select data from the suppliers table that you access using the database connection, conn. Assign the returned cursor object to the variable curs.

```
sqlquery = 'select City from suppliers';  
curs = exec(conn,sqlquery)
```

```
curs =
```

```
    Attributes: []  
           Data: 0  
DatabaseObject: [1x1 database]  
    RowLimit: 0  
    SQLQuery: 'select City from suppliers'  
    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 with the cursor object, close the cursor.

```
close(curs);
```

### **Select Data from a Database Table 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 returned cursor object to the variable curs.

```
sqlquery = 'select * from productTable';  
curs = exec(conn,sqlquery,10)
```

```
curs =
```

```
    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 with the cursor object, close the cursor.

```
close(curs);
```

### Select One Column of Data from the Database Table

Using dbtoolboxdemo data source, select stockNumber data from the productTable table that you access using the database connection conn. Assign the SQL statement to the variable sqlquery and assign the returned cursor to the variable curs.

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

After you are finished with the cursor object, close the cursor.

```
close(curs);
```

## Use a Variable in a Query

Using `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 MATLAB Command Window.

```
Train Set
```

To perform the query using your input, run the following code.

```
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 strings `select * from productTable where productDescription =` and `'productdesc'`. The pairs of four quotation marks are needed to create the pair of single quotation marks that appears in the SQL statement around `productdesc`. The outer two marks delineate the next string to concatenate, and two marks are required inside them to denote a quotation mark inside a string.

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 are finished with the cursor object, close the cursor.

```
close(curs);
```

### **Roll Back or Commit Data Exported to the Database Table**

Use `exec` to roll back or commit data after running a `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 don't 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 are finished with the cursor object, close the cursor.

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

Commit the data.

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

After you are finished with the cursor object, close the cursor.

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

### **Change the Database Connection Catalog**

Change the catalog for the database connection `conn` to `intlprice`.

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

After you are finished with the cursor object, close the cursor.

```
close(curs);
```

### **Create a Table and Add a New Column**

Use the SQL `CREATE` command to create the table.

```
sqlquery = ['CREATE TABLE Person(LastName varchar, '...  
          'FirstName varchar,Address varchar,Age int)'];
```

Create the table for the database connection object `conn`.

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

Use the SQL `ALTER` command to add a new column, `City`, to the table.

```
sqlquery = 'ALTER TABLE Person ADD City varchar(30)';  
curs = exec(conn,sqlquery);
```

After you are finished with the cursor object, close the cursor.

```
close(curs);
```

## Run a Stored Procedure and Return the Result Set

Use the JDBC interface to connect to a Microsoft SQL Server database, run a stored procedure, and return the result set. 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. The procedure definition is as follows.

```
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`.

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 =
```

```
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 more information, see database.

To return the result set in table format, use setdbprefs to set 'DataReturnFormat' to 'table'.

```
setdbprefs('DataReturnFormat','table');
```

Run 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 =
```

```
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 supplier data from `curs` using `fetch`.

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

```
curs =
```

```

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 running 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 Cursor Object, `curs`, and then close the connection Database Object, `conn`.

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

### **Run a Custom Database Function**

This example shows how to run a user-defined database function on Microsoft SQL Server.

Consider a database function, `get_prodCount`, that gets entry counts in a table, `productTable`.

```
CREATE FUNCTION dbo.get_prodCount()  
RETURNS int  
AS  
BEGIN  
    DECLARE @PROD_COUNT int  
    SELECT @PROD_COUNT = count(*) from productTable  
    RETURN(@PROD_COUNT)  
END  
GO
```

Use the database connection, `conn`, to execute the custom function from MATLAB.



```
sqlquery = 'SELECT dbo.get_prodCount() as num_products';  
curs = exec(conn,sqlquery);  
curs = fetch(curs);
```

After you are finished with the cursor object, close the cursor.

```
close(curs);
```

**See Also**

close | database | fastinsert | fetch | procedures |  
querybuilder | querytimeout | resultset | rsmd | set |  
update

**Concepts**

- “Using the exec Function” on page 4-39
- “Using the Native ODBC Database Connection” on page 2-12
- “Data Retrieval Restrictions” on page 1-7

# exportedkeys

---

**Purpose** 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
2	Schema containing primary key that is exported	SCOTT

Column	Description	Value
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/1.4.2/docs/api/java/sql/DatabaseMetaData>

## See Also

`crossreference` | `dmd` | `get` | `importedkeys` | `primarykeys`

# fastinsert

---

**Purpose** Add MATLAB data to database table

- Alternatives**
- Export data using Visual Query Builder with **Data operation** set to **Insert**.
  - Use the `datainsert` function. The `datainsert` function is faster than the `fastinsert` function, but you must enter dates and times in a specific format.

**Syntax** `fastinsert(conn, 'tablename', colnames, exdata)`

**Description** `fastinsert(conn, 'tablename', colnames, exdata)` exports records from the MATLAB variable `exdata` into new rows in an existing database table `tablename` via the connection `conn`. The variable `exdata` can be a cell array, numeric matrix, table, dataset, or structure. You do not specify the type of data you are exporting; the data is exported in its current MATLAB format. Specify column names for `tablename` as strings in the MATLAB cell array `colnames`. If `exdata` is a structure, field names in the structure must match `colnames`. If `exdata` is a table or a dataset, the variable names in the table or dataset must exactly match `colnames`.

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.

Use `update` to replace existing data in a database.

When working with a JDBC driver connection or a JDBC-ODBC bridge connection established using the `database` function, `fastinsert` offers improved performance over `insert`. This is because `insert` creates and executes an SQL insert query for each row of data. `fastinsert` creates the insert query only once and then allows for the data values to be plugged in. All rows of data get inserted as a batch resulting in an overall faster performance over `insert`. However, since `fastinsert`

relies more on driver functions compared to `insert`, it is possible in some edge case scenarios that the driver functions do not work as expected. In such cases, `insert` might be preferred, especially if the data to be inserted is small. `datainsert` is faster than `fastinsert` but needs data to be formatted in a specific way and accepts cell arrays and numeric matrices as input data.

When working with a native ODBC connection established using the `database.ODBCConnection` function, `fastinsert` and `insert` are identical. `datainsert` is not supported for native ODBC connections.

---

**Note** `fastinsert` supports the native ODBC interface. To insert dates and timestamps with the native ODBC interface, use the format 'YYYY-MM-DD HH:MM:SS.MS'.

---

## Tips

- The `fastinsert` function replaces the `insert` function. The two functions have the same syntax, but `fastinsert` provides better performance and supports more object types than `insert`. If `fastinsert` does not work as expected, try running `insert`.
- To reduce conversion time, convert dates to serial date numbers using `datenum` before calling `fastinsert`.
- To insert dates and timestamps with the native ODBC interface, use the format 'YYYY-MM-DD HH:MM:SS.MS'.
- To insert data into a structure, table, or dataset, use the following special formatting. Each field or variable in a structure, table or dataset must be a cell array or double vector of size  $m \times 1$ , where  $m$  is the number of rows to be inserted.
- The order of records in your database is not constant. Use values in column names to identify records.
- 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.

## Examples

### Example 1 – Insert a Table Record Using Native ODBC

- 1 Create a connection conn using the native ODBC interface and the dbtoolboxdemo data source.

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

```
conn =
```

```
ODBCConnection 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.

- 2 Select and display the data from the productTable.

```
curs = exec(conn, 'select * from productTable');  
curs = fetch(curs);  
curs.Data
```

```
ans =
```

```
productNumber    stockNumber    supplierNumber    unitCost    produ
```

9	125970	1003	13
8	212569	1001	5
7	389123	1007	16
2	400314	1002	9
4	400339	1008	21
1	400345	1001	14
5	400455	1005	3
6	400876	1004	8
3	400999	1009	17
10	888652	1006	24

- 3** Store the column names of `productTable` in a cell array.

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

- 4** Store the data for the insert in a cell array, `exdata`. The data contains `productNumber` equal to 11, `stockNumber` equal to 500565, `supplierNumber` equal to 1010, `unitCost` equal to \$20, and `productDescription` equal to 'Cooking Set'. Then, convert the cell array to a table, `exdata_table`.

```
exdata = {11, 500565, 1010, 20, 'Cooking Set'};
exdata_table = cell2table(exdata, 'VariableNames', colnames)
```

```
exdata_table =
```

productNumber	stockNumber	supplierNumber	unitCost
-----	-----	-----	-----
11	500565	1010	20

- 5** Insert the table data into the `productTable`.

```
fastinsert(conn, 'productTable', colnames, exdata_table);
```

- 6** Display the data from the `productTable` again.

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

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

```
ans =
```

productNumber	stockNumber	supplierNumber	unitCost	product
9	125970	1003	13	'Vict
8	212569	1001	5	'Trai
7	389123	1007	16	'Engi
2	400314	1002	9	'Pain
4	400339	1008	21	'Spac
1	400345	1001	14	'Buil
5	400455	1005	3	'Tin
6	400876	1004	8	'Sail
3	400999	1009	17	'Slin
10	888652	1006	24	'Tedo
11	500565	1010	20	'Cook

A new row appears in the productTable with the data from exdata\_table.

## Example 2 – Insert a Record

- 1 Using the dbtoolboxdemo data source, insert a record consisting of three columns, productNumber, Quantity, and Price into the inventoryTable. productNumber is 7777, Quantity is 100, and Price is 50.00. The database connection is conn. Assign the data to the cell array exdata.

```
exdata = {7777, 100, 50.00}
```

- 2 Create a cell array containing the column names in inventoryTable.

```
colnames = {'productNumber', 'Quantity', ' Price'}
```

- 3 Insert the data into the database.



```
fastinsert(conn, 'inventoryTable', colnames, exdata)
```

The row of data is added to the inventoryTable table.

### Example 3 – Insert Multiple Records

- Using the dbtoolboxdemo data source, insert multiple rows of data consisting of three columns, productNumber, Quantity, and Price into the inventoryTable.

```
exdata = {7778, 125, 23.00; 7779, 1160, 14.7; 7780, 150, 54.5}
```

- Create a cell array containing the column names in inventoryTable.

```
colnames = {'productNumber', 'Quantity', ' Price'}
```

- Insert the data into the database.

```
fastinsert(conn, 'inventoryTable', colnames, exdata)
```

The records are inserted into the table.

In addition, there are three 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*

### Example 4 – Import Records, Perform Calculations, and Export Data

This example shows how to retrieve sales data from a salesVolume table, calculate the sum of sales for 1 month, store this data in a cell array, and export this data to a yearlySales table.

- 1 Connect to the data source, dbtoolboxdemo, if needed.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see [database](#).

- 2 Use `setdbprefs` to set the format for retrieved data to numeric.

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

- 3 Import 10 rows of data the March column of data from the `salesVolume` table.

```
curs = exec(conn, 'select March from salesVolume');  
curs = fetch(curs);
```

- 4 Assign the data to the MATLAB workspace variable `AA`.

```
AA = curs.Data  
AA =
```

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

- 5 Calculate the sum of the March sales and assign the result to the variable `sumA`.

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

```
14606
```

- 6 Assign the month and sum of sales to a cell array to export to a database. Put the month in the first cell of `exdata`.

```
exdata(1,1) = {'March'}
exdata =
    'March'
```

Put the sum in the second cell of exdata.

```
exdata(1,2) = {sumA}
exdata =
    'March'    [14606]
```

- 7 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`.

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

- 8 Use the `get` function to determine the current 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 update; if it is on, data is automatically committed to the database.

```
get(conn, 'AutoCommit')
ans =
    on
```

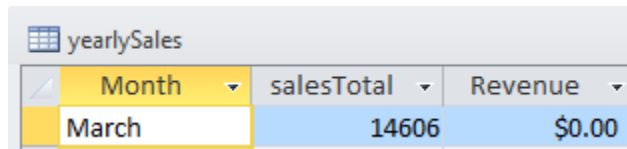
The `AutoCommit` flag is set to on, so the exported data is automatically committed to the database.

- 9 Use the `fastinsert` function to export the data into the `yearlySales` table. Pass the following arguments to this function.
  - `conn`, the connection object for the database
  - `yearlySales`, the name of the table to which you are exporting data
  - The cell arrays `colnames` and `exdata`

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

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

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



Month	salesTotal	Revenue
March	14606	\$0.00

- 11 Close the cursor.

```
close(curs);
```

## Example 5 – Insert Numeric Data

Using the dbtoolboxdemo data source, export exdata, a numeric matrix consisting of three columns, into the inventoryTable table.

```
exdata = [25, 439, 60.00]  
fastinsert(conn, 'inventoryTable', {'productNumber','Quantity', 'Price'}, exdata)
```

## Example 6 – Insert and Commit Data

- 1 Working with the dbtoolboxdemo data source, use the SQL commit function to commit data to a database after it has been inserted. The AutoCommit flag is off.

Insert the cell array exdata into the column names colnames of the inventoryTable table.

```
set(conn, 'AutoCommit', 'off')  
exdata = {157, 358, 740.00}  
colnames = {'productNumber', 'Quantity', ' Price'}  
fastinsert(conn, 'inventoryTable', colnames, exdata)  
commit(conn)
```

Alternatively, commit the data using an SQL commit statement with the `exec` function.

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

### Example 7 – Insert BOOLEAN Data

- 1 Using the `dbtoolboxdemo` data source, insert `BOOLEAN` data (which is represented as `MATLAB` type `logical`) into a database.

```
conn = database('dbtoolboxdemo', '', '');  
P.InvoiceNumber{1} = 2101;  
P.Paid{1}=logical(1);  
fastinsert(conn, 'Invoice', ...  
    {'InvoiceNumber'; 'Paid'}, P)
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see `database`.

- 2 View the new record in the database to verify that the `Paid` field is `BOOLEAN`. In some databases, the `MATLAB` logical value 0 is shown as a `BOOLEAN false`, `No`, or a cleared check box.

### See Also

`commit` | `database` | `exec` | `insert` | `logical` | `querybuilder` | `rollback` | `set` | `update`

### Tutorials

- “Getting Started with Visual Query Builder” on page 3-2

### How To

- “Using the Native ODBC Database Connection” on page 2-12

# fetch

---

## Purpose

Import data into MATLAB workspace from cursor object or from execution of SQL statement

## Syntax

```
curs = fetch(curs)
curs = fetch(curs,rowlimit)

results = fetch(conn,sqlquery)
results = fetch(conn,sqlquery,fetchbatchsize)
```

## 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`.

`results = fetch(conn,sqlquery)` executes the SQL statement `sqlquery`, imports all rows of data in batches for the open database connection `conn`, and returns the resulting data `results`.

`results = fetch(conn,sqlquery,fetchbatchsize)` imports all rows of data in batches of a specified number of rows `fetchbatchsize` at a time.

## Input Arguments

### **curs** - Database cursor

database cursor object

Database cursor, specified as an open SQL database cursor object created using `exec`.

### **conn** - Database connection

connection object

Database connection, specified as a database connection object created using `database`.

### **sqlquery** - SQL statement

SQL string

SQL statement, specified as an SQL string to execute.

#### **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.

#### **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 helps reduce overall retrieval time. If `fetchbatchsize` is not provided, a default value of 'FetchBatchSize' is used. 'FetchBatchSize' is set using `setdbprefs`.

#### **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.

#### **results - Result data**

cell array | table | data set | structure | numeric matrix

Result data, returned as a cell array, table, data set, structure, or numeric matrix as specified by 'DataReturnFormat' in `setdbprefs`.

## Examples

### Import All Rows of Data with the Native ODBC Interface Using the Cursor Object

Create a connection `conn` using the native ODBC interface and the `dbtoolboxdemo` data source.

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

```
conn =
```

```
ODBCConnection 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`, and store the data in a cell array contained in the cursor object field `curs.Data`.

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

```
curs =
```

```
ODBCCursor with properties:
```

```
Data: {10x1 cell}  
RowLimit: 0  
SQLQuery: 'select productDescription from productTable'  
Message: []
```



```
Type: 'ODBCursor Object'  
Statement: [1x1 database.internal.ODBCStatementHandle]
```

With the native ODBC interface, `curs` returns an `ODBCursor 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 are finished with the cursor object, close the cursor object.

```
close(curs);
```

### Import All Rows of Data Using the Cursor Object

Working with the `dbtoolboxdemo` data source, use `exec` to select data in column `City`, for example, in table `suppliers`. Then, use `fetch` to import all data from the SQL statement into the cursor object `curs`, and store the data in a cell array contained in the cursor object field `curs.Data`.

```
curs = exec(conn,'select City from suppliers');  
curs = fetch(curs)
```

```
curs =
```

```
Attributes: []
    Data: {10x1 cell}
DatabaseObject: [1x1 database]
    RowLimit: 0
    SQLQuery: 'select City from suppliers'
    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 =

    'New York'
    'London'
    'Adelaide'
    'Dublin'
    'Boston'
    'New York'
    'Wellesley'
    'Nashua'
    'London'
    'Belfast'
```

After you are finished with the cursor object, close the cursor object.

```
close(curs);
```

## **Import a Specified Number of Rows Using the Cursor Object**

Working with the `dbtoolboxdemo` data source, use the `rowlimit` argument to retrieve only the first three rows of data.

```
curs = exec(conn, 'select productdescription from producttable');
curs = fetch(curs, 3)
```

```
curs =
    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 are finished with the cursor object, close the cursor object.

```
close(curs);
```

## **Import Rows Iteratively Until You Retrieve All Data Using the Cursor Object**

Working with the `dbtoolboxdemo` data source, use the `rowlimit` argument to retrieve the first two rows of data, and then rerun the import using a `while` loop, retrieving two rows at a time. Continue until you have retrieved all data, which occurs when `curs.Data` is `'No Data'`.

```
curs = exec(conn, 'select productdescription from producttable');  
% 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 are finished with the cursor object, close the cursor object.

```
close(curs);
```

### **Import Numeric Data Using the Cursor Object**

Working with the dbtoolboxdemo data source, import a column of numeric data, using the setdbprefs function to specify numeric as the format for the retrieved data.

```
curs=exec(conn, 'select unitCost from productTable');  
setdbprefs('DataReturnFormat','numeric')  
curs=fetch(curs,3);  
curs.Data
```

```
ans =  
  
13  
5  
16
```

After you are finished with the cursor object, close the cursor object.

```
close(curs);
```

## Import BOOLEAN Data Using the Cursor Object

Import data that includes a BOOLEAN field, using the `setdbprefs` function to specify `cellarray` as the format for the retrieved data.

```
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 are finished with the cursor object, close the cursor object.

```
close(curs);
```

## Perform Incremental Fetch Using the Cursor Object

Working with the `dbtoolboxdemo` data source, retrieve data incrementally to avoid Java heap errors. Use `fetch` with the `setdbprefs` properties for `FetchInBatches` and `FetchBatchSize` to fetch large data sets.

```
setdbprefs('FetchInBatches', 'yes');
setdbprefs('FetchBatchSize', '2');
conn = database('dbtoolboxdemo', '', '');
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 fetch. 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 decided based on several factors such as the size per row being retrieved, the Java heap memory value, the driver's default fetch size, and system architecture, and hence, may vary from site to site.

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.

You can exercise 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.

After you are finished with the cursor object, close the cursor object.

```
close(curs);
```

## Import Data Using the Database Connection Object

`fetch` automatically imports data from the specified SQL statement when you pass a database object, `conn`, as the first argument. Use this example when using an ODBC/JDBC bridge or a JDBC interface. For the native ODBC interface, use `curs` as the input argument.

Using the `dbtoolboxdemo` data source that you access using the database connection object, `conn`, 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
```

### **Import Data with fetchbatchsize Using the Database Connection Object**

fetch automatically imports data from the specified SQL statement when you pass a database object, conn, as the first argument. Use this example when using an ODBC/JDBC bridge or a JDBC interface. For the native ODBC interface, use curs as the input argument.

Using the dbtoolboxdemo data source that you access using the database connection object, conn, import the productDescription column from the productTable by 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.

## **Import Two Columns of Data and View Information About the Data Using the Database Connection Object**

fetch automatically imports data from the specified SQL statement when you pass a database object, conn, as the first argument. Use this example when using an ODBC/JDBC bridge or a JDBC interface. For the native ODBC interface, use curs as the input argument.

Using the dbtoolboxdemo data source that you access using the database connection object, conn, import the InvoiceNumber and Paid columns from the Invoice table. Set the data return format to 'cellarray' using setdbprefs.

```
setdbprefs('DataReturnFormat','cellarray');  
sqlquery = 'select InvoiceNumber, Paid from Invoice';
```

```
results = fetch(conn, sqlquery);
```

View the size of the cell array into which the results were returned.

```
size(results)
```

```
ans =
```

```
    12     2
```

View the results for the first row of data.

```
results(1,:)
```

```
ans =
```

```
[2101] [0]
```

View the data type of the second element in the first row of data.

```
class(results{1,2})
```

```
ans =
```

```
logical
```

## See Also

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

## Related Examples

- “Retrieving Image Data Types” on page 4-25

## Concepts

- “Using the fetch Function” on page 4-42
- “Using the Native ODBC Database Connection” on page 2-12
- “Preference Settings for Large Data Import” on page 3-10
- “Data Retrieval Restrictions” on page 1-7

# fetchmulti

---

**Purpose** Import data from multiple resultsets

**Syntax** `curs = fetchmulti(curs)`

**Description** `curs = fetchmulti(curs)` imports data from the open SQL cursor object `curs` into the object `curs`, where the open SQL cursor object contains multiple resultsets.

Multiple resultsets are retrieved via `exec` with a `sqlquery` statement that runs a stored procedure consisting of two select statements.

`cursmulti.Data` contains data from each resultset associated with `cursmulti.Statement`. `cursmulti.Data` is a cell array consisting of cell arrays, structures, or numeric matrices as specified in `setdbprefs`; the data type is the same for all resultsets.

**Examples** Use `exec` to run a stored procedure that includes multiple select statements and `fetchmulti` to retrieve the resulting multiple resultsets.

```
conn = database(...)
setdbprefs('DataReturnFormat','cellarray')
curs = exec(conn, '{call sp_1}');
curs = fetchmulti(curs)
Attributes: []
          Data: {{10x1 cell} {12x4 cell}}
DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: '{call sp_1}'
          Message: []
          Type: 'Database Cursor Object'
ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
           [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
          Cursor: ...
[1x1 com.mathworks.toolbox.database.sqlExec]
          Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
           [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
          Fetch: ...
[1x1 com.mathworks.toolbox.database.fetchTheData]
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see `database`.

## See Also

`fetch` | `database` | `exec` | `setdbprefs`

# get

---

**Purpose** Retrieve object properties

**Syntax**

```
v = get(object)
v = get(object, 'property')
v.property
```

**Description** `v = get(object)` returns a structure that contains `object` and its corresponding properties, and assigns the structure to `v`.

`v = get(object, 'property')` retrieves the value of `property` for `object` and assigns the value to `v`.

`v.property` returns the value of `property` after you have created `v` by running `get`.

Use `set(object)` to view a list of writable properties for `object`.

Allowable objects include:

- “Database Connection Objects” on page 5-95, which are created using `database`
- “Cursor Objects” on page 5-96, which are created using `exec` or `fetch`
- “Driver Objects” on page 5-97, which are created using `driver`
- “Database Metadata Objects” on page 5-97, which are created using `dmd`
- “Drivermanager Objects” on page 5-97, which are created using `drivermanager`
- “ResultSet Objects” on page 5-98, which are created using `resultset`
- “ResultSet Metadata Objects” on page 5-98, which are created using `rsmd`

If you call these objects from applications that use Oracle Java, you can get more information about object properties from the Java Web site:

<http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/DatabaseMetaData.htm>

## Database Connection Objects

Allowable property names and returned values for database connection objects appear in the following table.

Property	Value
'AutoCommit'	Status of the AutoCommit flag. It is either on or off, as specified by set
'Catalog'	Name of the catalog in the data source. You may need to 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'	Identifies a JDBC connection object
'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'	Value for LoginTimeout
'TransactionIsolation'	Value of current transaction isolation mode
'Type'	Object type, specifically Database Object
'URL'	For JDBC connections only, the JDBC URL object <code>jdbc:subprotocol:subname</code> , as specified by database
'UserName'	User name required to connect to a given database, as specified by database
'Warnings'	Warnings returned by database

## Cursor Objects

Allowable property names and returned values for cursor objects appear in the following table.

Property	Value
'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'	Resultset object identifier
'Cursor'	Cursor object identifier
'Statement'	Statement object identifier  <hr/> <b>Note</b> If you specify a value (in seconds) for the timeout argument, queries time out after the time exceeds the given value. <hr/>
'Fetch'	0 for cursor created using exec; fetchTheData for cursor created using fetch



## Driver Objects

Allowable property names and examples of values for driver objects appear in the following table.

Property	Example of Value
'MajorVersion'	1
'MinorVersion'	1001

## Database Metadata Objects

Database metadata objects have many properties. Some allowable property names and examples of their values appear in the following table.

Property	Example of Value
'Catalogs'	{4x1 cell}
'DatabaseProductName'	'ACCESS'
'DatabaseProductVersion'	'03.50.0000'
'DriverName'	'JDBC-ODBC Bridge (odbcjt32.dll)'
'MaxColumnNameLength'	64
'MaxColumnsInOrderBy'	10
'URL'	'jdbc:odbc:dbtoolboxdemo'
'NullsAreSortedLow'	1

## Drivermanager Objects

Allowable property names and examples of values for drivermanager objects appear in the following table.

Property	Example of Value
'Drivers'	{'oracle.jdbc.driver.OracleDriver@1d8e09ef' [1x37 char]}
'LoginTimeout'	0
'LogStream'	[]

### ResultSet Objects

Allowable property names and examples of values for resultset objects appear in the following table.

Property	Example of Value
'CursorName'	{'SQL_CUR92535700x' 'SQL_CUR92535700x'}
'MetaData'	{1x2 cell}
'Warnings'	{{[] []}}

### ResultSet Metadata Objects

Allowable property names and examples of values for a resultset metadata objects appear in the following table.

Property	Example of Value
'CatalogName'	{'' ''}
'ColumnCount'	2
'ColumnName'	{'Calc_Date' 'Avg_Cost'}
'ColumnTypeName'	{'TEXT' 'LONG'}
'TableName'	{'' ''}
'isNullable'	{{[1] [1]}
'isReadOnly'	{{[0] [0]}

The empty strings for `CatalogName` and `TableName` indicate that databases do not return these values.

For command-line help on `get`, use the overloaded methods:

```
help cursor/get
help database/get
help dmd/get
help driver/get
help drivermanager/get
help resultset/get
help rsmd/get
```

## Examples

### Example 1 – Get Connection Property and Data Source Name

Connect to the database `dbtoolboxdemo`, and then get the name of the data source for the connection and assign it to `v`.

```
conn = database('dbtoolboxdemo', '', '');
v = get(conn, 'Instance')
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see `database`.

### Example 2 – Get Connection Property and AutoCommit Flag Status

Check the status of the `AutoCommit` flag for the database connection `conn`.

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

```
ans =
    on
```

### Example 3 – Display Data in Cursor

Display data in the cursor object `curs` by running:

```
curs = exec(conn, 'select productdescription from producttable')
```

```
curs = fetch(curs);  
get(curs, 'Data')
```

or:

```
curs.Data
```

```
ans =
```

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

## Example 4 – Get Database Metadata Object Properties

- 1 View the properties of the database metadata object for connection conn.

```
dbmeta = dmd(conn);  
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
```

**2** To view names of the catalogs in the database, run:

```
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'
```

## See Also

`columns` | `fetch` | `database` | `dmd` | `driver` | `drivermanager` | `exec` | `getdatasources` | `resultset` | `rows` | `rsmd` | `set`

## Purpose

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 strings. 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 more information 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'
```

## See Also

database | get | setdbprefs

# importedkeys

---

**Purpose** 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



Column	Description	Value
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/1.4.2/docs/api/java/sql/DatabaseMetaData>

# importedkeys

---

## **See Also**

[crossreference](#) | [dmd](#) | [exportedkeys](#) | [get](#) | [primarykeys](#)

**Purpose** 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
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

# indexinfo

---

Column	Description	Value
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 more information about the index information, see the `getIndexInfo` property on the Oracle Java Web site:

<http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/DatabaseMetaData.html>

## See Also

dmd | get | tables

**Purpose** Add MATLAB data to database tables

**Syntax** `insert(conn, 'tab', colnames, exdata)`

**Description** `insert(conn, 'tab', colnames, exdata)`

`insert` exports records from the MATLAB variable `exdata` into new rows in an existing database table `tablename` via the connection `conn`. The variable `exdata` can be a cell array, numeric matrix, table, dataset, or structure. You do not specify the type of data you are exporting; the data is exported in its current MATLAB format. Specify column names for `tablename` as strings in the MATLAB cell array `colnames`. If `exdata` is a structure, field names in the structure must exactly match `colnames`. If `exdata` is a table or a dataset, the variable names in the table or dataset must exactly match `colnames`.

When working with a JDBC driver connection or a JDBC-ODBC bridge connection established using the `database` function, `fastinsert` offers improved performance over `insert`. This is because `insert` creates and executes an SQL insert query for each row of data. `fastinsert` creates the insert query only once and then allows for the data values to be plugged in. All rows of data get inserted as a batch resulting in an overall faster performance over `insert`. However, since `fastinsert` relies more on driver functions compared to `insert`, it is possible in some edge case scenarios that the driver functions do not work as expected. In such cases, `insert` might be preferred, especially if the data to be inserted is small. `datainsert` is faster than `fastinsert` but needs data to be formatted in a specific way and accepts cell arrays and numeric matrices as input data.

When working with a native ODBC connection established using the `database.ODBCConnection` function, `fastinsert` and `insert` are identical. `datainsert` is not supported for native ODBC connections.

`insert` uses the same syntax as `fastinsert`.

---

**Notes:**

- `insert` supports the native ODBC interface. To insert dates and timestamps with the native ODBC interface, use the format 'YYYY-MM-DD HH:MM:SS.MS'.
  - To insert data into a structure, table, or dataset, use the following special formatting. Each field or variable in a structure, table, or dataset must be a cell array or double vector of size  $m \times 1$ , where  $m$  is the number of rows to be inserted.
- 

## Examples

### Example 1 – Insert a Table Record Using Native ODBC

- 1 Create a connection `conn` using the native ODBC interface and the `dbtoolboxdemo` data source.

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

```
conn =
```

```
ODBCConnection 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.

- 2 Select and display the data from the `productTable`.

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

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

```
ans =
```

productNumber	stockNumber	supplierNumber	unitCost
9	125970	1003	13
8	212569	1001	5
7	389123	1007	16
2	400314	1002	9
4	400339	1008	21
1	400345	1001	14
5	400455	1005	3
6	400876	1004	8
3	400999	1009	17
10	888652	1006	24

- 3** Store the column names of `productTable` in a cell array.

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

- 4** Store the data for the insert in a cell array, `exdata`. The data contains `productNumber` equal to 11, `stockNumber` equal to 400565, `supplierNumber` equal to 1010, `unitCost` equal to \$10, and `productDescription` equal to 'Rubik's Cube'. Then, convert the cell array to a table, `exdata_table`.

```
exdata = {11, 400565, 1010, 10, 'Rubik's Cube'};
exdata_table = cell2table(exdata, 'VariableNames', colnames)
```

```
exdata_table =
```

productNumber	stockNumber	supplierNumber	unitCost
11	400565	1010	10

# insert

---

- 5** Insert the table data into the productTable.

```
insert(conn, 'productTable', colnames, exdata_table);
```

- 6** Display the data from the productTable again.

```
curs = exec(conn, 'select * from productTable');  
curs = fetch(curs);  
curs.Data
```

```
ans =
```

productNumber	stockNumber	supplierNumber	unitCost	product
9	125970	1003	13	'Vict
8	212569	1001	5	'Trai
7	389123	1007	16	'Engi
2	400314	1002	9	'Pair
4	400339	1008	21	'Spac
1	400345	1001	14	'Buil
5	400455	1005	3	'Tin
6	400876	1004	8	'Sail
3	400999	1009	17	'Slin
10	888652	1006	24	'Tedo
11	400565	1010	10	'Rub

A new row appears in the productTable with the data from exdata\_table.

## Example 2— Insert the Contents of a Cell Array

- 1** Using the dbtoolboxdemo data source, select and display the data from the yearlySales table.

```
curs = exec(conn, 'select * from yearlySales');  
curs = fetch(curs);  
curs.Data
```

```
ans =
```



Month	salesTotal	Revenue
'January'	130	1200
'Feb'	25	250

- 2** Store the column names of `yearlySales` in a cell array.

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

- 3** Store the data for the insert in a cell array, `exdata`. The data contains `Month` equal to `'March'`, `salesTotal` equal to `$50`, and `Revenue` equal to `$2000`.

```
exdata = {'March',50,2000};
```

- 4** Insert the data into the `yearlySales` table.

```
insert(conn, 'yearlySales', colnames, exdata);
```

- 5** Display the data from the `yearlySales` table 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 the `yearlySales` table with the data from `exdata`.

## See Also

`commit` | `fastinsert` | `querybuilder` | `rollback`

## **How To**

- “Using the Native ODBC Database Connection” on page 2-12

**Purpose** Detect whether database connections are valid

**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`

# isdriver

---

**Purpose** Detect whether driver is valid JDBC driver object

**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** `driver` | `get` | `isjdbc` | `isurl`

**Purpose** Detect whether driver is JDBC compliant

**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** `driver` | `get` | `isdriver` | `isurl`

# isnullcolumn

---

**Purpose** Detect whether last record read in resultset is NULL

**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

[get](#) | [resultset](#)

# isreadonly

---

**Purpose** Detect whether 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)
```

The result indicates that the database connection `conn` is read only:

```
a =  
    1
```

Therefore, you cannot run `fastinsert`, `insert`, or `update` functions on this database.

**See Also** `database` | `isconnection`



---

<b>Purpose</b>	Detect whether database URL is valid
<b>Syntax</b>	<code>a = isurl(d, 's')</code>
<b>Description</b>	<code>a = isurl(d, 's')</code> returns 1 if the database URL <code>s</code> for the driver object <code>d</code> is valid. It returns 0 otherwise. The URL <code>s</code> is of the form <code>jdbc:odbc:name</code> or <code>name</code> .
<b>Examples</b>	Check whether the database URL <code>jdbc:odbc:thin:@144.212.123.24:1822:</code> is valid for driver object <code>d</code> . <pre>a = isurl(d, 'jdbc:odbc:thin:@144.212.123.24:1822:') a =     1</pre> <p>This indicates that the database URL is valid for <code>d</code>.</p>
<b>See Also</b>	<code>driver</code>   <code>get</code>   <code>isdriver</code>   <code>isjdbc</code>

# logintimeout

---

**Purpose** 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
```

# logintimeout

---

**3** Verify the timeout value.

```
logintimeout('oracle.jdbc.driver.OracleDriver')
ans =
     5
```

## **See Also**

database | get | set

**Purpose** Map resultset column name to resultset column index

**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 string or cell array of strings 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`

# ping

---

**Purpose** Get status information about database connection

**Syntax** ping(conn)

**Description** ping(conn) returns status information about the database connection conn if the connection is open. It returns an error message otherwise.

## **Examples**      **Example 1 – Get Status Information About ODBC Connection**

Check the status of the ODBC connection conn.

```
ping(conn)
ans =
    DatabaseProductName: 'ACCESS'
    DatabaseProductVersion: '03.50.0000'
    JDBCDriverName: 'JDBC-ODBC Bridge (odbcjt32.dll)'
    JDBCDriverVersion: '1.1001 (04.00.4202)'
    MaxDatabaseConnections: 64
    CurrentUserName: 'admin'
    DatabaseURL: 'jdbc:odbc:dbtoolboxdemo'
    AutoCommitTransactions: 'True'
```

## **Example 2 – Get Status Information About JDBC Connection**

Check the status of the JDBC connection conn.

```
ping(conn)
ans =
    DatabaseProductName: 'Oracle'
    DatabaseProductVersion: [1x166 char]
    JDBCDriverName: 'Oracle JDBC driver'
    JDBCDriverVersion: '7.3.4.0.2'
    MaxDatabaseConnections: 0
    CurrentUserName: 'scott'
    DatabaseURL: 'jdbc:oracle:thin: ...
@144.212.123.24:1822:orcl'AutoCommitTransactions: 'True'
```

---

### **Example 3 – Unsuccessful Request for Information About Connection**

In this example, the database connection conn has been terminated or is not successful.

```
ping(conn)
Cannot Ping the Database Connection
```

### **See Also**

database | dmd | get | isconnection | set | supports

# primarykeys

---

**Purpose** 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

[crossreference](#) | [dmd](#) | [exportedkeys](#) | [get](#) | [importedkeys](#)

# procedurecolumns

---

**Purpose** 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 more information about the procedurecolumns results, see the `getProcedureColumns` property on the Oracle Java Web site:

<http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/DatabaseMetaData>

## See Also

dmd | get | procedures

# procedures

---

**Purpose** 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(conn)
curs =
    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

# querybuilder

---

<b>Purpose</b>	Start SQL query builder GUI to import and export data
<b>Compatibility</b>	querybuilder is not recommended. Use dexplore instead.
<b>Syntax</b>	querybuilder
<b>Description</b>	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 more information on Visual Query Builder, including examples, see the VQB **Help** menu or “Getting Started with Visual Query Builder” on page 3-2.

**Purpose** 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
```

**Limitations**

- If a given database does not have a database timeout feature, it returns the following:

```
[Driver]Driver not capable
```

- ODBC drivers for Microsoft Access and Oracle do not support `querytimeout`.

**See Also** `exec`

# register

---

**Purpose** Load database driver

**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. The `register` function also allows you to run `drivermanager` with `set` and `get` on properties for loaded drivers.

**Examples** **1** `register(d)` loads the database driver object `d`.

**2** `get(d)` returns properties of the driver object.

**See Also** `driver` | `drivermanager` | `get` | `set` | `unregister`



**Purpose** Construct resultset object

**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 `clearwarnings`, `isnullcolumn`, and `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** `clearwarnings` | `close` | `fetch` | `exec` | `get` | `isnullcolumn` | `namecolumn` | `rsmd`

# rollback

---

**Purpose** Undo database changes

**Syntax** `rollback(conn)`

**Description** `rollback(conn)` reverses changes made to a database using `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** `rollback` does not roll back data in MySQL databases if the database engine is not `InnoDB`.

---

## 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`.

```
fastinsert(conn, 'DEPT', ...
{'DEPTNO'; 'DNAME'; 'LOC'}, exdata)
```

**3** Roll back the data that you inserted into the database by running:

```
rollback(conn)
```

The data in `exdata` is removed from the database. The database now contains the data it had before you ran the `fastinsert` function.

## See Also

`commit` | `database` | `exec` | `fastinsert` | `get` | `insert` | `update`

**Purpose** 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`, where `curs` has been generated by the `fetch` function.

**Examples** There are four rows in the fetched data set `curs`.

```
numrows = rows(curs)
```

```
numrows =  
    4
```

To see the four rows of data in `curs`, run:

```
curs.Data  
ans =  
    'Germany'  
    'Mexico'  
    'France'  
    'Canada'
```

**See Also** `cols` | `fetch` | `get` | `rsmd`

# rsmd

---

**Purpose** Construct resultset metadata object

**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`

## Purpose

Run SQL script on a database

## Syntax

```
results = runsqlscript(connect,sqlfilename)
results = runsqlscript(connect,sqlfilename,Name,Value)
```

## Description

`results = runsqlscript(connect,sqlfilename)` runs the SQL commands in the file `sqlfilename` on the connected database, and returns a cursor array.

`results = runsqlscript(connect,sqlfilename,Name,Value)` uses additional options specified by one or more `Name, Value` pairs.

## Input Arguments

### **connect - Database connection**

connection object

Database connection, specified as a connection object.

### **sqlfilename - File name of SQL commands**

string

File name of SQL commands to run, specified as a string. The file must be a text file, and can contain comments along with SQL queries. Single line comments must start with `--`. Multiline comments should be wrapped in `/*...*/`.

**Example:** `'C:\work\sql_file.sql'`

### **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 time out**

0 implies unlimited time (default) | positive scalar

Query time out (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 Mth 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 the data is the result of long and complex SQL queries that are difficult to convert into MATLAB strings. runsqlscript is not designed to handle SQL scripts containing continuous PL/SQL blocks with BEGIN and END, such as stored procedure definitions, trigger definitions, and so on. However, table definitions do work.

- An SQL script containing any of the following can produce unexpected results:
  - Apostrophes that are not escaped (including those in comments). For example, the string 'Here's the code' should be written as 'Here''s the code'.
  - Nested comments.

## Examples

### Run SQL Script

Run SQL commands from a file on a connected data source.

To run this example, set up the data source, `dbtoolboxdemo`, by following the steps in “Set Up the `dbtoolboxdemo` 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 directory.

Create the connection object to the data source, `dbtoolboxdemo`.

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

User names and passwords are not required for this connection.

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 =
```

```
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 run this example, set up the data source, dbtoolboxdemo, by following the steps in “Set Up the dbtoolboxdemo 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 directory.

Create the connection object to the data source, dbtoolboxdemo.

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

User names and passwords are not required for this connection.

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see `database`.

Run the SQL script, `compare_sales.sql`, specifying 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 run this example, set up the data source, `dbtoolboxdemo`, by following the steps in “Set Up the `dbtoolboxdemo` 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 directory.

Create the connection object to the data source, `dbtoolboxdemo`.

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

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see `database`.

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. The optimal value for `FetchBatchSize` is decided based on several factors like the size per row being retrieved, the Java heap memory value, the driver's default fetch size, and system architecture, and hence, may vary from site to site. For more information on estimating a value for `FetchBatchSize`, see “Preference Settings for Large Data Import” on page 3-10.

```
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.

## 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.

# runsqlscript

---

## Definitions

### Batch

One or more SQL statements terminated by either a semicolon or the keyword GO.

## See Also

resultset | setdbprefs | fetch

## Related Examples

- “Set Up the dbtoolboxdemo Data Source”

## Concepts

- “Preference Settings for Large Data Import” on page 3-10

## Purpose

Call stored procedure with input and output parameters

## Syntax

```
results = runstoredprocedure(conn, sp_name, parms_in,  
                             types_out)
```

## Description

`results = runstoredprocedure(conn, sp_name, parms_in, types_out)` calls a stored procedure with specified input parameters and returns output parameters, for the database connection handle `conn`. `sp_name` is the stored procedure to run, `parms_in` is a cell array containing the input parameters for the stored procedure, and `types_out` is the list of data types for the output parameters.

Use `runstoredprocedure` to return the value of a variable to a MATLAB variable, which you cannot do when running a stored procedure via `exec`. Running a stored procedure via `exec` returns resultsets but cannot return output parameters.

## Examples

These examples illustrate how `runstoredprocedure` differs from running stored procedures via `exec`.

- 1 Run a stored procedure that has no input or output parameters:

```
x = runstoredprocedure(c, 'myprocnoparams')
```

- 2 Run a stored procedure given input parameters 2500 and 'Jones' with no output parameters.

```
x = runstoredprocedure(c, 'myprocinonly', {2500, 'Jones'})
```

- 3 Run the stored procedure `myproc` given input parameters 2500 and 'Jones'. Return an output parameter of type `java.sql.Types.NUMERIC`, which could be any numeric Oracle Java data type. The output parameter `x` is the value of a database variable `n`. The stored procedure `myproc` creates this variable, given the input values 2500 and 'Jones'. For example, `myproc` computes `n`, the number of days when Jones is 2500. It then returns the value of `n` to `x`.

# runstoredprocedure

---

```
x = runstoredprocedure(c, 'myproc', {2500, 'Jones'}, {java.sql.Types.NUMERIC})
```

## See Also

[fetch](#) | [exec](#)

**Purpose**

Set properties for database, cursor, or drivermanager object

**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 5-152, created using `database`
- “Cursor Objects” on page 5-153, created using `exec` or `fetch`
- “Drivermanager Objects” on page 5-153, created using `drivermanager`

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.

## 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'	Database data is written and automatically committed when you run <code>fastinsert</code> , <code>insert</code> , or <code>exec</code> . You cannot use <code>rollback</code> to reverse this process.
	'off'	Database data is not committed automatically when you run <code>fastinsert</code> , <code>insert</code> , or <code>update</code> . Use <code>rollback</code> to reverse this process. When you are sure that your data is correct, use the <code>commit</code> function to commit it to the database.
'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 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.

## Drivermanager Objects

The allowable *property* and value for a drivermanager object appear in the following table.

Property	Value	Description
'LoginTimeout'	positive integer	Sets the logintimeout value for all loaded database drivers.

For command-line help on set, use the overloaded methods:

```
help cursor/set
help database/set
help drivermanager/set
```

## 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 =
    Attributes: []
           Data: {5x8 cell}
DatabaseObject: [1x1 database]
      RowLimit: 5
    SQLQuery: 'select * from EMP'
      Message: []
           Type: 'Database Cursor Object'
    ResultSet: [1x1 oracle.jdbc.driver.OracleResultSet]
           Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
    Statement: [1x1 oracle.jdbc.driver.OracleStatement]
           Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

The RowLimit property of curs is 5 and the Data property is 5x8 cell, indicating that fetch returned five rows of data.

In this example, RowLimit limits the maximum number of rows you can retrieve. Therefore, rerunning the fetch function returns no data.

## Example 2 – Set the AutoCommit Flag to On

This example shows what happens when you run a database update function 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`.

```
fastinsert(conn, 'Growth', colnames, exdata)
```

The data is inserted and committed to the database.

## **Example 3 – Set the AutoCommit Flag to Off and Commit Data**

This example shows the results of running `fastinsert` 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`.

```
fastinsert(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 insert data into 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)
```

- 3 Roll back the data.

```
rollback(conn)
```

---

The data in the table is now as it was before you ran update.

### **Example 5 – Set the LoginTimeout for a DriverManager Object**

- 1** Create a drivermanager object dm and set its LoginTimeout value to 3 seconds.

```
dm = drivermanager;  
set(dm, 'LoginTimeout', 3);
```

- 2** Verify this result.

```
logintimeout  
ans =  
    3
```

### **See Also**

fetch | database | drivermanager | exec | fastinsert | get |  
insert | logintimeout | ping | update

# setdbprefs

---

<b>Purpose</b>	Set preferences for retrieval format, errors, NULLs, and more
<b>Alternative</b>	<ul style="list-style-type: none"><li>From the Database Explorer Toolstrip, select <b>Preferences</b> to open the Database Toolbox Preferences dialog box.</li></ul>
<b>Syntax</b>	<pre>setdbprefs s = setdbprefs setdbprefs('property') setdbprefs('property', 'value') setdbprefs({'property1'; ...}, {'value1'; ...}) setdbprefs(s)</pre>
<b>Description</b>	<p>setdbprefs returns current values for database preferences.</p> <p>s = setdbprefs returns current values for database preferences to the structure s.</p> <p>setdbprefs('property') returns the current value for the specified property.</p> <p>setdbprefs('property', 'value') sets the specified property to value.</p> <p>setdbprefs({'property1'; ...}, {'value1'; ...}) sets properties starting with property1 to values starting with value1.</p> <p>setdbprefs(s) sets preferences specified in the structure s to values that you specify.</p> <p>Allowable properties appear in the following tables:</p> <ul style="list-style-type: none"><li>DataReturnFormat and ErrorHandling Properties and Values for setdbprefs on page 5-159</li><li>Null Data Handling Properties and Values for setdbprefs on page 5-160</li><li>Other Properties and Values for setdbprefs (Not Accessible via Query &gt; Preferences) on page 5-162</li></ul>

**DataReturnFormat and ErrorHandling Properties and Values for setdbprefs**

Property	Allowable Values	Description
'DataReturnFormat'	'cellarray' (default), 'table', 'dataset', 'numeric', or 'structure'	Format for data to import into the MATLAB workspace. Set the format based on the type of data being retrieved, memory considerations, and your preferred method of working with retrieved data.
	'cellarray' (default)	Import nonnumeric data into MATLAB cell arrays.
	'table'	Import data into MATLAB table objects. Use for all data types. Facilitates working with returned columns.
	'dataset'	Import data into MATLAB dataset objects. Use for all data types. Facilitates working with returned columns. This option requires Statistics Toolbox.
	'numeric'	Import data into 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.

# setdbprefs

## DataReturnFormat and ErrorHandling Properties and Values for setdbprefs (Continued)

Property	Allowable Values	Description
'ErrorHandling'	'store' (default), 'report', or 'empty'	Specify how to handle errors when importing data. Set this parameter before you run <code>exec</code> .
	'store' (default)	Errors from running <code>database</code> are stored in the <code>Message</code> field of the returned connection object. Errors from running <code>exec</code> are stored in the <code>Message</code> field of the returned cursor object.
	'report'	Errors from running <code>database</code> or <code>exec</code> appear immediately in the MATLAB Command Window.
	'empty'	Errors from running <code>database</code> are stored in the <code>Message</code> field of the returned connection object. Errors from running <code>exec</code> are stored in the <code>Message</code> field of the returned cursor object. Objects that cannot be created are returned as empty handles ( <code>[]</code> ).

## Null Data Handling Properties and Values for setdbprefs

Property	Allowable Values	Description
'NullNumberRead'	User-specified, for example, '0'	Specify how NULL numbers appear after being imported from a database into the MATLAB workspace. NaN is the default value. String values such as 'NULL' cannot be set if



**Null Data Handling Properties and Values for setdbprefs (Continued)**

<b>Property</b>	<b>Allowable Values</b>	<b>Description</b>
		'DataReturnFormat' is set to 'numeric'. Set this parameter before running fetch.
'NullNumberWrite'	User-specified, for example, 'NaN' (default)	Numbers in the specified format, for example, NaN appears as NULL after being exported from the MATLAB workspace to a database.
'NullStringRead'	User-specified, for example, 'null' (default)	Specify how NULL strings appear after being imported from a database into the MATLAB workspace. Set this parameter before running fetch.
'NullStringWrite'	User-specified, for example, 'null' (default)	Strings in the specified format, for example, 'NULL', appear as NULL after being exported from the MATLAB workspace to a database.

# setdbprefs

## Other Properties and Values for setdbprefs (Not Accessible via Query > Preferences)

Property	Allowable Values	Description
'JDBCDataSourceFile'	User-specified, for example, 'D:/file.mat'	Path to MAT-file containing JDBC data sources. For more information, see "Accessing Existing JDBC Data Sources" on page 2-4.
'UseRegistryForSources'	'yes' (default) or 'no'	When set to yes, VQB searches the Microsoft Windows registry for ODBC data sources that are not uncovered in the system ODBC.INI file. The following message might appear: Registry editing has been disabled by your administrator. This message is harmless and can safely be ignored.
'TempDirForRegistryOutput'	User-specified, for example, 'D:/work'	Folder where VQB writes ODBC registry settings when you run <code>getdatasources</code> . Use when you add data sources and do not have write access to the MATLAB Current Folder. The default is the Windows temporary folder, which is returned by the command <code>getenv('temp')</code> .  If you specify a folder to which you do not have write access or which does not exist, the following error appears:  Cannot export <folder-name>\ODBC.INI: Error opening the file. There may be a disk or file system error.

**Other Properties and Values for setdbprefs (Not Accessible via Query > Preferences) (Continued)**

Property	Allowable Values	Description
'DefaultRowPreFetch'	User-specified numeric value, default value is '10000'	Number of rows fetched from the Database server at a time for any query. The higher the number, the fewer the number of trips to the server. This setting is applicable only for databases that allow this number to be set, e.g, Oracle.
'FetchInBatches'	'yes' or 'no' (default)	Automates fetching in batches for large data sets where you might 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'	User-specified numeric value, default value is '1000'. Supported values are from 1000 to 1000000.	Automates fetching in batches for large data sets when used in conjunction 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 more information on estimating a 'FetchBatchSize' value, see "Preference Settings for Large Data Import" on page 3-10.

## Tips

- Preferences are retained across MATLAB sessions.
- Regardless of the value of 'NullNumberWrite', a NULL value is always written to the database when you input [] or NaN for a numeric data type.
- For string inputs, a NULL value is written to the database only when the input value matches the value of 'NullStringWrite'.

## Examples

### Example 1 – Display Current Values

Run setdbprefs.

```
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'
```

These values show that:

- Data is imported from databases into MATLAB cell arrays.
- Errors that occur during a database connection or SQL query attempt are stored in the `Message` field of the connection or cursor data object.
- Each NULL number in the database is read into the MATLAB workspace as NaN. Each NaN in the MATLAB workspace is exported to the database as NULL. Each NULL string in the database is read into the MATLAB workspace as 'null'. Each 'null' string in the MATLAB workspace is exported to the database as a NULL string.

- A MAT-file that specifies the JDBC source file has not been created.
- Visual Query Builder looks in the Windows system registry for data sources that do not appear in the ODBC.INI file.
- No temporary folder for registry settings has been specified.
- The default number of rows fetched from the Database server at a time for any query is 10,000.

## Example 2 – Change a Preference

- 1 Run `setdbprefs ('NullNumberRead')`.

```
setdbprefs ('NullNumberRead')
NullNumberRead: 'NaN'
```

Each NULL number in the database is read into the MATLAB workspace as NaN.

- 2 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.

## Example 3 – Change the DataReturnFormat Preference

- 1 Specify that database data be imported into MATLAB cell arrays.

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

- 2 Import data into the MATLAB workspace.

```
conn = database('dbtoolboxdemo', '', '');
curs=exec(conn, ...
'select productnumber,productdescription from producttable');
curs=fetch(curs,3);
curs.Data
```

```
ans =  
  
    [9]    'Victorian Doll'  
    [8]    'Train Set'  
    [7]    'Engine Kit'
```

Alternatively, you can use the native ODBC interface for an ODBC connection. For more information, see `database`.

- 3 Change the data return format from `cellarray` to `numeric`.

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

- 4 Perform the same import operation as you ran in the cell array example. Note the format of the returned data.

```
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, the MATLAB software treats them as NULL numbers and assigns them the current value for the `NullNumberRead` property of `setdbprefs`, `NaN`.

- 5 Change the data return format to `structure`.

```
setdbprefs ('DataReturnFormat','structure')
```

- 6 Perform the same import operation as you ran in the cell array example.

```
curs.Data  
  
ans =  
  
    productnumber: [3x1 double]  
 productdescription: {3x1 cell}
```

- 7 View the contents of the structure to see the data.

```
curs.Data.productdescription  
  
ans =  
  
    'Victorian Doll'  
    'Train Set'  
    'Engine Kit'
```

```
curs.Data.productnumber  
  
ans =  
  
    9  
    8  
    7
```

#### Example 4 – Change the Write Format for NULL Numbers

- 1 Specify NaN for the NullNumberWrite format.

```
setdbprefs('NullNumberWrite', 'NaN')  
  
colnames = {'productNumber' , 'Quantity', 'Price'}
```

Numbers represented as NaN in the MATLAB workspace are exported to databases as NULL.

For example, the variable `ex_data` contains a NaN.

```
ex_data = {24, NaN, 30.00}
```

- 2 Insert `ex_data` into a database using `fastinsert`. The NaN data is exported into the database as NULL.

```
fastinsert (conn, 'inventoryTable', colnames, ex_data)
```

## Example 5 – Specify Error Handling Settings

- 1 Specify the store format for the `ErrorHandling` preference.

```
setdbprefs ('ErrorHandling','store')
```

Errors generated from running `database` or `exec` are stored in the `Message` field of the returned connection or cursor object.

- 2 Try to fetch data from a closed cursor.

```
conn=database('dbtoolboxdemo', '', '');
curs=exec(conn, 'select productdescription from producttable');
close(curs)
curs=fetch(curs,3);

curs =

    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.

- 3 To specify the report format for the ErrorHandling preference, run:

```
setdbprefs ('ErrorHandling','report')
```

Errors generated by running database or exec appear immediately in the Command Window.

- 4 Try to fetch data from a closed cursor.

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

- 5 Specify the empty format for the ErrorHandling preference.

```
setdbprefs ('ErrorHandling','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, [].

- 6 Try to fetch data from a closed cursor.

```
conn = database('dbtoolboxdemo', '', '');  
curs=exec(conn, 'select productdescription from producttable');
```

```
close(curs)
curs=fetch(curs,3);

curs =

    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 appears in the cursor object `Message` field. Furthermore, the `Attributes` field contains empty handles because no attributes could be created.

## Example 6 – Change Multiple Settings

Specify that NULL strings are read from the database into a MATLAB matrix of doubles as 'NaN':

```
setdbprefs({'NullStringRead';'DataReturnFormat'},...
{'NaN';'numeric'})
```

See “Example 8 — Assign Values to a Structure” on page 5-171 for more information on another way to change multiple settings.

## Example 7 – Specify JDBC Data Sources for Use by VQB

Instruct VQB to connect to the database using the data sources specified in the file `myjdbcdatasources.mat`.

```
setdbprefs('JDBCDataSourceFile',...
'D:/Work/myjdbcdatasources.mat')
```

## Example 8 – Assign Values to a Structure

- 1 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'
```

- 2 Set preferences using the values in s:

```
setdbprefs(s)
```

- 3 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'
```

## Example 9 – Return Values to a Structure

- 1 Assign values for all preferences to s by running:

```
s = setdbprefs
s =
```

# setdbprefs

---

```
DataReturnFormat: 'cellarray'  
  ErrorHandling: 'store'  
  NullNumberRead: 'NaN'  
  NullNumberWrite: 'NaN'  
  NullStringRead: 'null'  
  NullStringWrite: 'null'  
  JDBCDataSourceFile: ''  
  UseRegistryForSources: 'yes'  
  TempDirForRegistryOutput: 'C:\Work'  
  DefaultRowPreFetch: '10000'  
  FetchInBatches: 'no'  
  FetchBatchSize: '1000'
```

- 2 Use the MATLAB tab completion feature when obtaining the value for a preference. For example, enter:

```
s.U
```

- 3 Press the **Tab** key, and then **Enter**. MATLAB completes the field and displays the value.

```
s.UseRegistryForSources
```

```
ans =
```

```
yes
```

## Example 10 – Save Preferences

You can save your preferences to a MAT-file to use them in future MATLAB sessions. For example, say that you need to reuse preferences that you set for the Seasonal Smoothing project. Assign the preferences to the variable `SeasonalSmoothing` and save them to a MAT-file `SeasonalSmoothingPrefs` in your current folder:

```
SeasonalSmoothing = setdbprefs;  
save SeasonalSmoothingPrefs.mat SeasonalSmoothing
```

Later, load the data and restore the preferences:

```
load SeasonalSmoothingPrefs.mat  
setdbprefs(SeasonalSmoothing);
```

## See Also

[clear](#) | [fetch](#) | [getdatasources](#)

## Related Examples

- “Preference Settings for Large Data Import” on page 3-10
- “Working with Preferences” on page 3-6

# sql2native

---

<b>Purpose</b>	Convert JDBC SQL grammar to SQL grammar native to system
<b>Syntax</b>	<code>n = sql2native(conn, 'sqlquery')</code>
<b>Description</b>	<code>n = sql2native(conn, 'sqlquery')</code> converts the SQL statement string <code>sqlquery</code> from JDBC SQL grammar into the database system's native SQL grammar for the connection <code>conn</code> . The native SQL statement is assigned to <code>n</code> .

**Purpose** Detect whether property is supported by database metadata object

**Syntax**

```
a = supports(dbmeta)
a = supports(dbmeta, 'property')
a.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.

`a.property` returns the value of `property` after you have created `a` using the `supports` function.

**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 See the value of the `GroupBy` property by running:

```
a.GroupBy
a =
    1
```

**See Also** `database` | `dmd` | `get` | `ping`

# tableprivileges

---

**Purpose** 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`



**Purpose** Return database table names

**Syntax**

```
t = tables(dbmeta, 'cata')
t = tables(dbmeta, 'cata', 'sch')
```

**Description** `t = tables(dbmeta, 'cata')` returns a list of tables and table types in the catalog `cata` for the database whose database metadata object is `dbmeta` resulting from a database connection object.

`t = tables(dbmeta, 'cata', 'sch')` returns a list of tables and table types in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta` resulting from a database connection object.

---

**Tip** For command-line help on `tables`, use the overloaded method:

```
help dmd/tables
```

---

**Examples** Get the table names and types for the schema `SCOTT` in the catalog `orcl`, for the database metadata object `dbmeta`.

```
t = tables(dbmeta, 'orcl', 'SCOTT')
t =
    'BONUS'      'TABLE'
    'DEPT'       'TABLE'
    'EMP'        'TABLE'
    'SALGRADE'  'TABLE'
    'TRIAL'     'TABLE'
```

**See Also** `attr` | `bestrowid` | `dmd` | `get` | `indexinfo` | `tableprivileges`

# unregister

---

<b>Purpose</b>	Unload database driver
<b>Syntax</b>	<code>unregister(d)</code>
<b>Description</b>	<code>unregister(d)</code> unloads the database driver object <code>d</code> , freeing up system resources. If you do not unload a registered driver, it automatically unloads when you end your MATLAB session.
<b>Examples</b>	<code>unregister(d)</code> unloads the database driver object <code>d</code> .
<b>See Also</b>	<code>register</code>

**Purpose**

Replace data in database table with MATLAB data

**Syntax**

```
update(conn, 'tab', colnames, exdata, 'whereclause')
update(conn, 'tab', colnames, ...
{datA,datAA, ...; datB,datBB, ...; datn, datNN}, ...
{'where col1 = val1'; 'where col2 = val2'; ... 'where coln = valn'})
```

**Description**

`update(conn, 'tab', colnames, exdata, 'whereclause')` exports the MATLAB variable `exdata` in its current format into the database table `tab` using the database connection `conn`. `exdata` can be a cell array, numeric matrix, or structure. Existing records in the database table are replaced as specified by the SQL `whereclause` command.

Specify column names for `tab` as strings in the MATLAB cell array `colnames`. If `exdata` is a structure, field names in the structure must match field names in `colnames`.

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 a SQL commit statement via the `exec` function. Roll back the data by running `rollback` or a SQL rollback statement via the `exec` function.

To add new rows instead of replacing existing data, use `fastinsert`.

`update(conn, 'tab', colnames, {dataA, dataAA, ...; datB, datBB, ...; datn,datNN}, {'where col1 = val1'; 'where col2 = val2'; ... 'where coln = valn'})` exports multiple records for `n` where clauses. The number of records in `exdata` must equal `n`.

**Tips**

- The order of records in your database is not constant. Use values of column names to identify records.
- An error like the following may appear if 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 repeat the `update` function.

- An error like the following may appear if you try to run an update operation that is identical to one that you just ran:

```
??? Error using ==> database.update
Error:Commit/Rollback Problems
```

## Examples

### Example 1 – Update an Existing Record

Using `dbtoolboxdemo` data source, update a record in the `inventoryTable` table using the database connection `conn`, where `productNumber` is 1, replacing the current value for `Quantity` with 2000.

- 1 Define a cell array containing the column name that you are updating, `Quantity`.

```
colnames = {'Quantity'}
```

- 2 Define a cell array containing the new data, 2000.

```
exdata(1,1) = {2000}
```

- 3 Run the update.

```
update(conn, 'inventorytable', colnames, exdata, ...
        'where productNumber = 1')
```

## Example 2 – Roll Back Data After Updating a Record

Using `dbtoolboxdemo` data source, update the column `Price` in the `inventoryTable` table for the record selected by `whereclause`, using data contained in the cell array `exdata`. The `AutoCommit` flag is off. The data is rolled back after the update operation is run.

- 1 Set the `AutoCommit` flag to off for database connection `conn`.

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

- 2 Define a cell array containing the new data, 30.00.

```
exdata(1,1) = {30.00}
```

- 3 Define a where clause.

```
whereclause = 'where productNumber = 1'
```

- 4 Update the `Price` column.

```
update(conn, 'inventoryTable', {'Price'}, exdata, whereclause)
```

- 5 Because the data was not committed, you can roll it back.

```
rollback(conn)
```

The update is reversed; the data in the table is the same as it was before you ran update.

## Example 3 – Update Multiple Records with Different Constraints

Using `dbtoolboxdemo` data source, given the table `inventoryTable`, where column names are `'productNumber'`, `'Quantity'`, and `'Price'`:

```
A = 10000
```

```
B = 5000
```

# update

---

Assign productNumbers with values of 5 and 8 to have a new Quantity with the values of A and B:

```
update(conn, 'inventoryTable', {'Quantity'}, {A;B}, ...  
{'where productNumber = 5';'where productNumber = 8'})
```

## **See Also**

commit | database | fastinsert | rollback | set

## Purpose

Automatically update table columns

## Syntax

```
v1 = versioncolumns(dbmeta, 'cata')  
v1 = versioncolumns(dbmeta, 'cata', 'sch')  
v1 = versioncolumns(dbmeta, 'cata', 'sch', 'tab')
```

## Description

`v1 = 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.

`v1 = 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.

`v1 = 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`.

```
v1 = versioncolumns(dbmeta, 'orcl', 'SCOTT', 'BONUS')  
v1 =  
    {}
```

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`

# width

---

**Purpose** 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 `cursor`.

**Examples** Get the width of the first column of the fetched data set, `cursor`:

```
colsize = width(cursor, 1)
```

```
colsize =
```

```
11
```

The field size of column one is 11 characters (bytes).

**See Also** `attr` | `cols` | `columnnames` | `fetch` | `get`



## A

- advanced query options in VQB 3-27
- All option in VQB 3-27
- arrays
  - data format 5-158
  - data format in VQB 3-9
- attr 5-2
- Attributes 5-96
- attributes of data
  - attr function 5-2
- AutoCommit
  - setting status 5-152
  - status via get 5-95

## B

- bestrowid 5-4
- BINARY data types
  - retrieving with functions 4-25
  - retrieving with VQB 3-51
- BOOLEAN data type
  - retrieving 5-24 5-86
  - VQB 3-54

## C

- Catalog 5-95
- CatalogName 5-98
- cell arrays
  - data format 5-158
  - for query results 4-4
  - setting data format in VQB 3-6
- charting
  - query results 3-19
- Charting dialog box 3-19
- clearwarnings 5-5
- close 5-6
- cols 5-8
- ColumnCount 5-98
- ColumnName 5-98

- columnnames 5-9
  - exporting example 4-15
- columnprivileges 5-10
- columns 5-11
  - automatically updated 5-183
  - cross reference 5-14
  - exported keys 5-66
  - foreign key information 5-104
  - imported key information 5-104
  - names, via attr 5-2
  - names, via columnnames 5-9
  - names, via columns 5-11
  - number 5-8
  - optimal set to identify row 5-4
  - primary key information 5-128
  - privileges 5-10
  - width 5-184
- ColumnTypeName 5-98
- columnWidth 5-2
- commit 5-12
- Condition in VQB 3-29
- confds
  - function reference 5-13
- Configure Data Source dialog box 5-13
- connection
  - clearing warnings for 5-5
  - close function 5-6
  - database, opening (establishing),
    - example 4-3
  - information 5-126
  - JDBC 5-95
  - messages 5-95
  - object 4-3
  - properties, getting 5-94
  - properties, setting 5-151
  - read only 5-120
  - status 5-126
  - status, example 4-3
  - time allowed for 5-122
  - time allowed for, example 4-3

- validity 5-115
- warnings 5-95
- constructor functions 4-36
- crossreference 5-14
- currency 5-2
- Current clauses area in VQB
  - example 3-30
- cursor
  - attributes 5-96
  - close function 5-6
  - creating via fetch 5-17
  - data element 5-96
  - error messages 5-96
  - objects
    - example 4-3
  - opening 4-3
  - properties 5-151
  - properties, example 5-94
  - resultset object 5-137
- Cursor 5-96
- cursor.fetch 5-17

## D

- data
  - attributes 5-2
  - column names 5-9
  - column numbers 5-8
  - commit function 5-12
  - committing 5-152
  - displaying results in VQB 3-15
  - exporting 5-68 5-109
  - field names 5-9
  - importing 5-17
  - information about 4-5
  - inserting into database 4-17
  - replacing 4-12
  - rolling back 5-138
  - rolling back, via set 5-152
  - rows function 5-139
  - unique occurrences of 3-27
  - updating 5-179
- Data 5-96
- Data Explorer
  - starting 5-49
- data format 5-158
  - Database Toolbox 3-9
  - preferences for retrieval 5-158
  - preferences in VQB 3-6
- data sources
  - defining
    - JDBC 5-13
  - JDBC
    - accessing 2-4
    - modifying 2-5
    - removing 2-6
    - updating 2-5
  - ODBC connection 5-95
  - ODBC, on system 5-103
- data types 5-2
  - BINARY, retrieving with functions 4-25
  - BINARY, retrieving with VQB 3-51
  - OTHER, retrieving with functions 4-25
  - OTHER, retrieving with VQB 3-51
  - supported 1-5
- database
  - connecting to, example 4-3
  - example 4-3
  - JDBC connection 5-95
  - metadata objects
    - creating 5-50
    - properties 5-94
    - properties supported 5-175
  - supported 1-2
  - Database Toolbox requirements 1-2
- database.fetch 5-41
- DatabaseObject 5-96
- dbdemos 4-1
- demos 4-1
  - dbinfodemo 4-5

- dbinsertdemo 4-8
- dbupdatedemo 4-12
- dexplore 5-49
- displaying
  - query results
    - as chart 3-19
    - as report 3-21
    - in MATLAB Report Generator software 3-22
    - relationally 3-15
- Distinct option in VQB 3-27
- dmd 5-50
  - example 4-27
- driver 5-51
  - example 4-34
  - object in get function 5-95
- driver objects
  - functions, example 4-34
  - properties 4-34
- drivermanager 5-52
- drivermanager objects
  - example 4-34
  - properties 5-151
  - properties, via get 5-94
- drivers
  - JDBC 1-3
    - troubleshooting 2-7
  - JDBC compliance 5-117
  - loading 5-136
  - ODBC 1-3
  - properties 5-94
  - properties, drivermanager 5-52
  - supported 1-3
  - unloading 5-178
  - validity 5-116
- Drivers 5-98

**E**

- editing clauses in VQB 3-31

- empty field 4-26
- error handling
  - preferences 3-6
- error messages
  - cursor object 5-96
  - database connection object 5-95
- error notification, preferences 5-158
- examples
  - using functions 4-1
- exec
  - example 4-3
  - with fetch 5-41
- exportedkeys 5-66
- exporting data
  - inserting 5-68 5-109
    - example 4-8
    - multiple records 4-17
  - replacing 5-179
  - replacing, example 4-12

**F**

- fastinsert 5-68
- fetch
  - cursor 5-17
  - database 5-41
- Fetch 5-96
- fetchbatchsize
  - database.fetch 5-41
- fetchmulti 5-92
- fieldName 5-2
- fields
  - names 5-11
  - size (width) 5-2
    - width 5-184
- foreign key information
  - crossreference 5-14
  - exportedkeys 5-66
  - importedkeys 5-104
- format for data retrieved, preferences 5-158

freeing up resources 5-6

functions

    equivalent to VQB queries 3-59

## G

get 4-35 5-94

    properties 4-34

getdatasources 5-103

grouping statements 3-32

    removing 3-36

## H

Handle 5-95

HAVING Clauses dialog box 3-39

Having in VQB 3-39

HTML report of query results 3-21

    MATLAB Report Generator software 3-22

## I

images

    importing 4-25

        VQB 3-51

importedkeys 5-104

importing data

    bulk insert

        example 4-18

    data types

        BINARY and OTHER using functions 4-25

        BINARY and OTHER using VQB 3-51

    empty field 4-26

    using functions 5-17

        example 4-3

index for resultset column 5-125

indexinfo 5-107

insert 5-109

inserting data into database 4-17

Instance 5-95

isconnection 5-115

isdriver 4-35 5-116

isjdbc 5-117

isNullable 5-98

isnullcolumn 5-118

isreadonly 5-120

isReadOnly 5-98

isurl 5-121

## J

Java® Database Connectivity. *See* JDBC  
JDBC

    compliance 5-117

    connection object 5-95

    driver instance 5-95

    drivers

        supported 1-3

        validity 5-116

    MAT-file location preference 5-158

    SQL conversion to native grammar 5-174

    URL

        via get 5-95

join operation in VQB 3-47

## L

logical data types

    retrieving 5-24 5-86

    VQB 3-54

logintimeout 5-122

    example 4-3

    Macintosh platform support 5-122

LoginTimeout

    Database connection object 5-95

    Drivermanager objects 5-98

    example 4-35

LogStream 5-98

## M

MajorVersion 5-97

- MATLAB Report Generator software
  - display of query results 3-22
- memory problems
  - fetchbatchsize solution 5-41
  - RowLimit solution 5-17
- Message
  - attr 5-2
  - cursor object 5-96
  - database connection object 5-95
- metadata objects
  - database 5-50
    - example 4-27
  - resultset 5-140
  - resultset functions 4-33
- methods 4-36
- MinorVersion 5-97

**N**

- namecolumn 5-125
- nested SQL 3-42
- NULL values
  - detecting in imported record 5-118
  - preferences for reading and writing 3-6
  - reading from database 4-14
  - representation in results 3-8
  - setdbprefs 5-158
  - writing to database 3-6
- nullable 5-2
- numeric data format 5-158
  - VQB 3-6

**O**

- objects 4-36
  - creating 4-36
  - properties, getting 5-94
- ObjectType 5-95
- ODBC
  - data sources on system 5-103

- drivers 1-3
- Open Database Connectivity. *See* ODBC
- Operator in VQB 3-31
- ORDER BY Clauses dialog box 3-37
- Order by option in VQB 3-36
- OTHER data types
  - retrieving with functions 4-25
  - retrieving with VQB 3-51

**P**

- parentheses, adding to statements 3-32
- ping 5-126
  - example 4-3
- platforms 1-2
- precision 5-2
- preferences
  - for Visual Query Builder 3-6
- primary key information 5-14
- primarykeys 5-128
- privileges
  - columns 5-10
  - tables 5-176
- procedurecolumns 5-130
- procedures 5-132
- properties
  - database metadata objects 5-175
    - example 4-28
  - drivers 4-34
  - getting 5-94
  - setting 5-151

**Q**

- queries
  - accessing subqueries in multiple tables 3-42
  - accessing values in multiple tables 3-47
  - displaying results
    - as chart 3-19
    - as report 3-21

- in MATLAB Report Generator
  - software 3-22
  - relationally 3-15
- ordering results 3-36
- refining 3-29
- results 5-96
- querybuilder 5-134
- querytimeout 5-135
- quotation marks
  - in table and column names 1-7

**R**

- readonly 5-2
- ReadOnly 5-95
- refining queries 3-29
- register 5-136
- Relation in VQB 3-29
- relational display of query results 3-15
- replacing data 4-12
  - update function 5-179
- reporting query results
  - MATLAB Report Generator software 3-22
  - table 3-21
- reserved words
  - in table and column names 1-7
- resultset 5-137
  - clearing warnings for 5-5
  - closing 5-6
  - column name and index 5-125
  - metadata objects 4-33
    - creating 5-140
    - properties 5-94
  - properties 5-94
- ResultSet 5-96
- retrieving data
  - restrictions 1-7
- rollback 5-138
- RowLimit
  - get 5-96

- set 5-153
- rows 5-139
  - uniquely identifying 5-4
- rsmd 5-140
- runstoredprocedure 5-149

## S

- scale 5-2
- set 5-151
  - example 4-35
- setdbprefs 5-158
  - example 4-14
- size 4-16
- Sort key number in VQB 3-37
- Sort order in VQB 3-37
- spaces
  - in table and column names 1-7
- speed
  - inserting data 5-68
- SQL
  - commands 1-4
  - conversion to native grammar 5-174
  - join in VQB 3-47
  - statement
    - in exec 5-96
    - in exec, example 4-3
    - in VQB 3-31
    - time allowed for query 5-135
    - where clause 5-179
  - sql2native 5-174
- SQLQuery 5-96
- Statement 5-96
- status of connection 5-126
  - example 4-3
- stored procedures
  - in catalog or schema 5-132
  - information 5-130
- string and numeric data format 5-158
- structure data format 5-158

- VQB 3-6
- subqueries
  - in VQB 3-42
- Subquery dialog box 3-43
- supports 5-175
  - example 4-31
- system requirements 1-2

## T

- TableName 5-98
- tableprivileges 5-176
- tables 5-177
  - example 4-33
  - index information 5-107
  - names 5-177
  - privileges 5-176
  - selecting multiple for VQB 3-48
- time
  - allowed for connection 5-122
  - allowed for SQL query 5-135
- Timeout 5-95
- TransactionIsolation 5-95
- Type 5-96
- typeName 5-2
- typeValue 5-2

## U

- ungrouping statements 3-36

- unique occurrences of data 3-27
- unregister 5-178
- update 5-179
  - example 4-12
- URL 5-95
  - validity 5-121
- user name 5-95

## V

- versioncolumns 5-183
- Visual Query Builder
  - advanced query options 3-27
  - equivalent Database Toolbox functions 3-59
  - getting started 3-2
  - starting 5-134
  - steps to export (insert) data 3-4
  - steps to import (retrieve) data 3-2
- VQB. *See* Visual Query Builder

## W

- Warnings 5-95
  - warnings, clearing 5-5
- where clause 5-179
- WHERE Clauses dialog box 3-29
- Where option in VQB 3-29
- width 5-184
- writable 5-95