

# Database Toolbox

For Use with MATLAB®

Computation  
|

Visualization  
|

Programming  
|

User's Guide  
*Version 2*



## How to Contact The MathWorks:



www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup



support@mathworks.com	Technical support
suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
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	Mail
--	------

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

### *Database Toolbox User's Guide*

© COPYRIGHT 1998 - 2002 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 or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:	May 1998	Online only	New for Version 1 for MATLAB 5.2
	July 1998	First printing	For Version 1
	June 1999	Online only	Updated for Version 2 (Release 11)
	December 1999	Second printing	For Version 2 (Release 11)
	September 2000	Online only	Updated for Version 2.1 (Release 12)
	June 2001	Third printing	Updated for Version 2.2 (Release 12.1)
	July 2002	Online only	Updated for Version 2.2.1 (Release 13)

## Preface

---

<b>What Is the Database Toolbox?</b> .....	<b>vi</b>
How Databases Connect to MATLAB .....	<b>vi</b>
Features of the Database Toolbox .....	<b>vii</b>
<b>Related Products</b> .....	<b>viii</b>
<b>Using This Guide</b> .....	<b>x</b>
Expected Background .....	<b>x</b>
Organization of the Document .....	<b>x</b>
Online Help .....	<b>xi</b>
<b>Typographical Conventions</b> .....	<b>xii</b>

## Setup

---

**1** |

<b>System Requirements</b> .....	<b>1-2</b>
Platforms .....	<b>1-2</b>
MATLAB Version .....	<b>1-2</b>
Databases .....	<b>1-3</b>
Drivers .....	<b>1-4</b>
Structured Query Language (SQL) .....	<b>1-4</b>
Data Types .....	<b>1-5</b>
<b>Installing the Database Toolbox</b> .....	<b>1-6</b>

<b>Setting Up a Data Source</b> .....	<b>1-7</b>
Setting Up a Data Source for ODBC Drivers .....	<b>1-7</b>
Setting Up a Data Source for JDBC Drivers .....	<b>1-11</b>
<b>Starting the Database Toolbox</b> .....	<b>1-13</b>

## Visual Query Builder

# 2

<b>Getting Started with the Visual Query Builder</b> .....	<b>2-2</b>
Before You Start .....	<b>2-2</b>
Starting the Visual Query Builder .....	<b>2-2</b>
Summary of Steps to Use the Visual Query Builder .....	<b>2-3</b>
Quitting the Visual Query Builder .....	<b>2-4</b>
When to Use the Visual Query Builder .....	<b>2-4</b>
When to Use Database Toolbox Functions .....	<b>2-5</b>
Examples for the Visual Query Builder .....	<b>2-5</b>
Online Help for the Visual Query Builder .....	<b>2-6</b>
<b>Creating, Running, and Saving a Query</b> .....	<b>2-7</b>
Building and Executing a Query .....	<b>2-7</b>
Saving a Query .....	<b>2-10</b>
Specifying Preferences for NULLS, Data Format, and Error Handling .....	<b>2-11</b>
Using Retrieved Data in MATLAB .....	<b>2-15</b>
Clearing Variables from the Data Area .....	<b>2-15</b>
<b>Viewing Query Results</b> .....	<b>2-16</b>
Relational Display of Data .....	<b>2-16</b>
Chart Display of Results .....	<b>2-20</b>
Report Display of Results in a Table .....	<b>2-23</b>
Customized Display of Results in the Report Generator .....	<b>2-25</b>
<b>Fine-Tuning Queries Using Advanced Query Options</b> ...	<b>2-27</b>
Retrieving Unique Occurrences .....	<b>2-27</b>
Retrieving Information That Meets Specified Criteria .....	<b>2-29</b>
Presenting Results in Specified Order .....	<b>2-38</b>

Creating Subqueries for Values from Multiple Tables . . . . .	2-42
Creating Queries for Results from Multiple Tables . . . . .	2-47
Other Features in Advanced Query Options . . . . .	2-51

## **Using Functions in the Database Toolbox**

# **3**

<b>Importing Data into MATLAB from a Database . . . . .</b>	<b>3-2</b>
<b>Viewing Information About the Imported Data . . . . .</b>	<b>3-8</b>
<b>Exporting Data from MATLAB to a New Record in a Database . . . . .</b>	<b>3-11</b>
<b>Replacing Existing Data in a Database MATLAB . . . . .</b>	<b>3-17</b>
<b>Exporting Multiple New Records from MATLAB . . . . .</b>	<b>3-19</b>
<b>Accessing Metadata . . . . .</b>	<b>3-23</b>
Resultset Metadata Object . . . . .	3-29
<b>Performing Driver Functions . . . . .</b>	<b>3-30</b>
<b>About Objects and Methods for the Database Toolbox . . .</b>	<b>3-33</b>
<b>Working with Cell Arrays in MATLAB . . . . .</b>	<b>3-36</b>
Viewing Cell Array Data Returned from a Query . . . . .	3-37
Viewing Elements of Cell Array Data . . . . .	3-39
Performing Functions on Cell Array Data . . . . .	3-41
Creating Cell Arrays for Exporting Data from MATLAB . . . .	3-41

<b>Functions—By Category</b> .....	<b>4-2</b>
General .....	<b>4-3</b>
Database Connection .....	<b>4-3</b>
SQL Cursor .....	<b>4-3</b>
Importing Data into MATLAB from a Database .....	<b>4-4</b>
Exporting Data from MATLAB to a Database .....	<b>4-4</b>
Database Metadata Object .....	<b>4-5</b>
Driver Object .....	<b>4-6</b>
Drivermanager Object .....	<b>4-6</b>
ResultSet Object .....	<b>4-6</b>
ResultSet Metadata Object .....	<b>4-6</b>
Visual Query Builder .....	<b>4-7</b>
<b>Functions—Alphabetical List</b> .....	<b>4-8</b>

# Preface

---

What Is the Database Toolbox? (p. vi)	Overview of how databases connect to MATLAB, toolbox functions and the Visual Query Builder, and the major features of the toolbox.
Related Products (p. viii)	Other toolboxes especially relevant to the Database Toolbox.
Using This Guide (p. x)	Expected background of users, organization of the document, and accessing online help.
Typographical Conventions (p. xii)	Conventions, especially those for portraying syntax.

## What Is the Database Toolbox?

The Database Toolbox is one of an extensive collection of toolboxes for use with MATLAB®. The Database Toolbox enables you to move data (both importing and exporting) between MATLAB and popular relational databases.

With the Database Toolbox, you can bring data from an existing database into MATLAB, use any of the MATLAB computational and analytic tools, and store the results back in the database or in another database. You read from the database, importing the data into the MATLAB workspace.

For example, a financial analyst working on a mutual fund could import a company's financial data into MATLAB, run selected analyses, and store the results for future tracking. The analyst could then export the saved results to a database.

## How Databases Connect to MATLAB

The Database Toolbox connects MATLAB to a database using MATLAB functions. Data is retrieved from the database and stored in the MATLAB workspace. At that point, you use the extensive set of MATLAB tools to work with the data. You can include Database Toolbox functions in MATLAB M-files. To export the data from MATLAB to a database, you use MATLAB functions.

The Visual Query Builder (VQB), which comes with the Database Toolbox, is an easy-to-use graphical user interface for retrieving data from your database. With the VQB, you build queries to retrieve data by selecting information from lists rather than by entering MATLAB functions. The VQB retrieves the data into the MATLAB workspace so you then can process the data using the MATLAB suite of functions. With the VQB, you can display the retrieved information in relational tables, reports, and charts.



## Features of the Database Toolbox

The Database Toolbox has the following features:

- Different databases can be used in a single session—Import data from one database, perform calculations, and export the modified or unmodified data to another database. Multiple databases can be open during a session.
- Data types are automatically preserved in MATLAB—No data massaging or manipulation is required. The data is stored in MATLAB as cell arrays or structures, which support mixed data types, or as numeric matrices, per your specification. Export numeric, cell array, or structure data.
- Retrieval of large data sets or partial data sets—You can retrieve large data sets from a database in a single fetch or in discrete amounts using multiple fetches.
- Retrieval of database metadata—You do not need to know the table names, field names, and properties of the database structure to access the database, but can retrieve that information using Database Toolbox metadata functions.
- Dynamic importing of data from within MATLAB—Modify your SQL queries in MATLAB statements to retrieve the data you need.
- Single environment for faster data analysis—Access both database data and MATLAB functions at the MATLAB command prompt.
- Multiple cursors supported for a single database connection—Once a connection has been established with a database, the connection can support the use of multiple cursors. You can execute several queries on the same connection.
- Export query results using the Report Generator—If the Report Generator product is installed locally, you can create custom reports from the Visual Query Builder.
- Database connections remain open until explicitly closed—Once the connection to a database has been established, it remains open during the entire MATLAB session until you explicitly close it. This improves database access and reduces the number of functions necessary to import/export data.
- Visual Query Builder—If you are unfamiliar with SQL, you can retrieve information from databases via this easy-to-use graphical interface.

## Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the Database Toolbox.

For more information about any of these products, see either

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at <http://www.mathworks.com>; see the “products” section

---

**Note** The toolboxes listed below all include functions that extend the capabilities of MATLAB.

---

<b>Product</b>	<b>Description</b>
Data Acquisition Toolbox	Acquire and send out data from plug-in data acquisition boards
Datafeed Toolbox	Acquire real-time financial data from data service providers
Financial Time Series Toolbox	Tool for analyzing time series data in the financial markets
Financial Toolbox	Analyze and manage financial time series data
GARCH Toolbox	Analyze financial volatility using univariate GARCH models
MATLAB Report Generator	Automatically generate documentation for MATLAB applications and data
MATLAB Runtime Server	Deploy runtime versions of MATLAB applications

Note that you cannot compile Database Toolbox functions into stand-alone executables using the MATLAB Compiler and the C/C++ Math Library. The Database Toolbox makes extensive use of MATLAB object-oriented programming and Java, neither of which is supported by the MATLAB Compiler.

## Using This Guide

This user's guide describes how to install and use the Database Toolbox.

### Expected Background

This user's guide assumes that you have a working understanding of MATLAB.

If you are not familiar with the Structured Query Language (SQL) and database applications, use the Visual Query Builder. For information on using the Visual Query Builder, see "Visual Query Builder" on page 2-1.

If you are familiar with SQL and the database applications you use, you can use the Visual Query Builder to build SQL queries easily and import results into MATLAB. If you want to export results from MATLAB to databases, write MATLAB applications that access databases, or perform functions not available with the Visual Query Builder, use the Database Toolbox functions. For information on how to use the functions, see "Using Functions in the Database Toolbox" on page 3-1, and "Function Reference" on page 4-1.

### Organization of the Document

The remainder of the book provides instructions for setting up and using the Database Toolbox.

Section	Description
Chapter 1, "Setup"	Provides system requirements and describes how to install the Database Toolbox and set up a data source for ODBC and JDBC drivers.
Chapter 2, "Visual Query Builder"	Provides instructions for using the Visual Query Builder, an easy-to-use graphical user interface for querying your database. It uses a sample database, dbtoolboxdemo, that is installed with the Database Toolbox for use with the U.S. English version of Microsoft Access 2000. If you have this version of Microsoft Access installed on your system, you can perform the steps exactly as shown.

<b>Section</b>	<b>Description (Continued)</b>
Chapter 3, “Using Functions in the Database Toolbox”	Presents examples with instructions for using many of the Database Toolbox functions. The examples use a sample database, Northwind, that is available from Microsoft for Access users. If you have Microsoft Access installed on your system, you can perform the steps exactly as shown. Another example uses a different database, tutorial, a database that is installed with the Database Toolbox for use with Access.
Chapter 4, “Function Reference”	A reference of all functions in the toolbox, with a summary presented by category and the details organized alphabetically.

## Online Help

- Help for the Database Toolbox is available online via the Help browser.
- Use the doc function for information about a specific function.
- In the Visual Query Builder, use the **Help** menu, or use the **Help** buttons in dialog boxes for detailed information about features in the dialog boxes.

## Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	Monospace font	To assign the value 5 to A, enter <code>A = 5</code>
Function names, syntax, filenames, directory/folder names, and user input	Monospace font	The <code>cos</code> function finds the cosine of each array element. Syntax line example is <code>MLGetVar ML_var_name</code>
Buttons and keys	<b>Boldface</b> with book title caps	Press the <b>Enter</b> key.
Literal strings (in syntax descriptions in reference chapters)	<b>Monospace bold</b> for literals	<code>f = freqspace(n, 'whole')</code>
Mathematical expressions	<i>Italics</i> for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$ .
MATLAB output	Monospace font	MATLAB responds with <code>A =</code> <code>5</code>
Menu and dialog box titles	<b>Boldface</b> with book title caps	Choose the <b>File Options</b> menu.
New terms and for emphasis	<i>Italics</i>	An <i>array</i> is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	<code>[c,ia,ib] = union(...)</code>
String variables (from a finite list)	<i>Monospace italics</i>	<code>sysc = d2c(sysd, 'method')</code>

# Setup

---

- |  |  |
|--|--|
| System Requirements (p. 1-2)             | Supported platforms, MATLAB versions, databases, drivers, SQL standard, and data types.                                |
| Installing the Database Toolbox (p. 1-6) | Follow standard installation instructions for MATLAB and toolboxes on your platform.                                   |
| Setting Up a Data Source (p. 1-7)        | Before connecting to a database, set up the data source, either local or remote for ODBC drivers, or for JDBC drivers. |
| Starting the Database Toolbox (p. 1-13)  | Start using functions or the Visual Query Builder.   |

## System Requirements

The Database Toolbox 2.2.1 works with the following systems and applications:

- “Platforms” on page 1-2
- “MATLAB Version” on page 1-2
- “Databases” on page 1-3
- “Drivers” on page 1-4
- “Structured Query Language (SQL)” on page 1-4
- “Data Types” on page 1-5

### Platforms

The Database Toolbox 2.2.1 runs on all of the platforms that support MATLAB Release 13 and Java. The Database Toolbox 2.2.1 does not run on the Hewlett-Packard 10.2 platform.

### MATLAB Version

The Database Toolbox 2.2.1 requires MATLAB Version 6.5 (Release 13) or later. You can see the system requirements for MATLAB online at <http://www.mathworks.com/products/sysreq/>.



## Databases

Your system must have access to an installed database. The Database Toolbox supports import/export of data from any ODBC/JDBC-compliant database management system, including the following.

- IBM DB2
- Informix
- Ingres
- Microsoft Access
- Microsoft SQL Server
- Oracle
- Sybase SQL Server
- Sybase SQL Anywhere

If you are upgrading from an earlier version of a database, such as Microsoft SQL Server Version 6.5, to a newer version, there is nothing special you need to do for the Database Toolbox. Just be sure to configure the data sources for the new version of the database application as you did for the original version.

---

**Note** We recommend that you do *not* include spaces in table and column names. Although Access supports the use of spaces in table and column names, most other databases do not. To retrieve data from tables whose names contain spaces, use delimiters around the table name when building the query. For example, some databases use brackets, [ ]. The Visual Query Builder does not allow spaces in table names.

Also, be sure not to name columns using the database's reserved words, such as DATE in Microsoft Access, or you will not be able to import or export the data using the Database Toolbox.

---

## Drivers

For Windows platforms, the Database Toolbox supports Open Database Connectivity (ODBC) drivers used with the supported databases. For UNIX and Windows platforms, the Database Toolbox supports Java Database Connectivity (JDBC) drivers. If you use JDBC drivers on a Windows platform, you cannot use the Visual Query Builder.

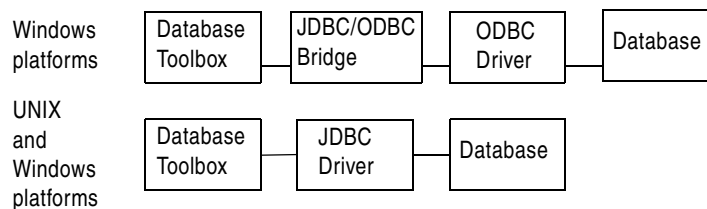
The driver for your database must be installed in order to use the Database Toolbox. Most users (or their database administrators) install the driver when they install the database. Consult your database documentation if you need instructions to install a database driver. If your database does not ship with JDBC drivers, you can download drivers from the Sun JDBC web site, <http://industry.java.sun.com/products/jdbc/drivers>.

### About Drivers for the Database Toolbox

An ODBC driver is a standard PC interface that enables communication between database management systems and SQL-based applications. A JDBC driver is a standard interface that enables communication between Java-based applications and database management systems.

The Database Toolbox is a Java-based application. To connect the Database Toolbox to a database's ODBC driver, the toolbox uses a JDBC/ODBC bridge, which is supplied and automatically installed as part of the toolbox.

The following illustrates the use of drivers with the Database Toolbox.



## Structured Query Language (SQL)

The Database Toolbox supports American National Standards Institute (ANSI) standard SQL commands.

## Data Types

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

- BOOLEAN
- CHAR
- DATE
- DECIMAL
- DOUBLE
- FLOAT
- INTEGER
- LONGCHAR (This is called the Memo data type in Microsoft Access.)
- NUMERIC
- REAL
- SMALLINT
- TIME
- TIMESTAMP
- TINYINT
- VARCHAR

Any other type of data that is *imported* is treated as a VARCHAR by MATLAB. If you import a data type that cannot be treated as a VARCHAR, you see an unsupported data message from MATLAB.

If you try to *export* MATLAB data types not on this list, you see a syntax error from the database.

## **Installing the Database Toolbox**

To install the Database Toolbox, select it with any other MATLAB toolboxes you want to install when you install MATLAB. For more information, see the installation documentation for your platform.

After installing a new version of the Database Toolbox, you need to set up your data sources. See “Setting Up a Data Source” on page 1-7.

## Setting Up a Data Source

Before you can connect from the Database Toolbox to a database, you need to set up a *data source*. A data source consists of data that you want the toolbox to access, and information about how to find the data, such as driver, directory, server, or network names. You assign a name to each data source.

The instructions for setting up a data source differ depending on your configuration. Use one of these sets of instructions:

- For MATLAB Windows platforms whose database resides on that PC or on another system to which the PC is networked via ODBC drivers, see “Setting Up a Data Source for ODBC Drivers” below.
- For MATLAB platforms that connect to a database via a JDBC driver, see “Setting Up a Data Source for JDBC Drivers” on page 1-11.

### Setting Up a Data Source for ODBC Drivers

Follow this procedure to set up a data source for a PC running Windows whose database resides on that PC or on another system to which the PC is networked via ODBC drivers. This procedure uses as an example, the Microsoft ODBC driver Version 4.00 and the U.S. English version of Microsoft Access 2000 for Windows 2000. If you have a different configuration, you may have to modify the instructions.

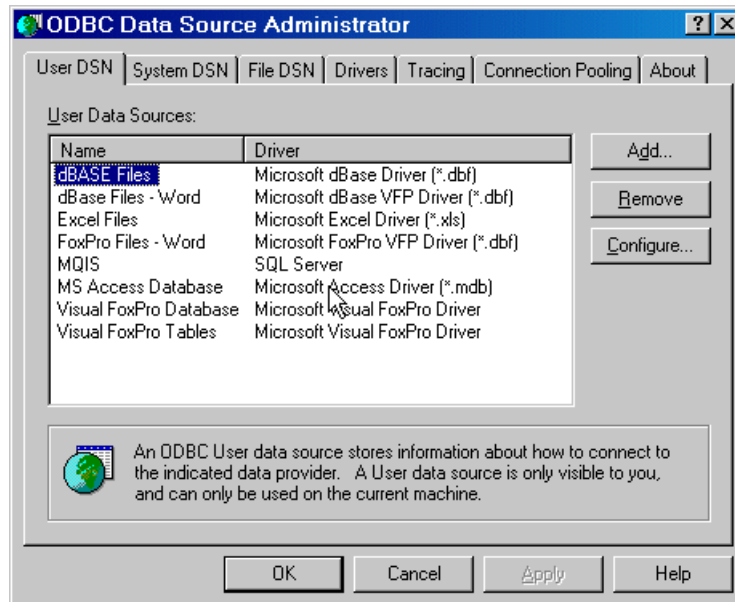
If you have Microsoft Access installed and want to use many of the examples in this document as written, set up these two data sources:

- `dbtoolboxdemo` data source—Uses the tutorial database provided with the Database Toolbox in `$matlabroot\toolbox\database\dbdemos`. When you first open the tutorial database (`tutorial.mdb`) in Access 2000, convert it to the Access 2000 format and assign it a different name, for example, `northwind.mdb`.
- `SampleDB` data source—Uses the Microsoft Access sample database called `northwind`. If you do not already have the sample database on your system, you can download it from the Microsoft Web site downloads page. It is part of the Access 2000 downloads and is the Northwind Traders sample database. When you first open the Northwind database (`Nwind.mdb`) in Access 2000, convert it to the Access 2000 format and save it. You may want to make a copy of the original unconverted tutorial database.

To set up the data source:

- 1 From the Windows **Start** menu, select **Programs -> Administrative Tools -> Data Sources (ODBC)**.

The **ODBC Data Source Administrator** dialog box appears, listing any existing data sources.



- 2 Select the **User DSN** tab.

A list of existing user data sources appears.

- 3 Click **Add**. A list of installed ODBC drivers appears in the **Create New Data Source** dialog box.

- 4 Select the ODBC driver that the local data source you are creating will use and click **Finish**.
  - For the examples in this book, select **Microsoft Access Driver (\*.mdb)**.
  - Otherwise, select the driver for your database.

The **ODBC Setup** dialog box appears for the driver you selected. Note that the dialog box for your driver might be different from the following.

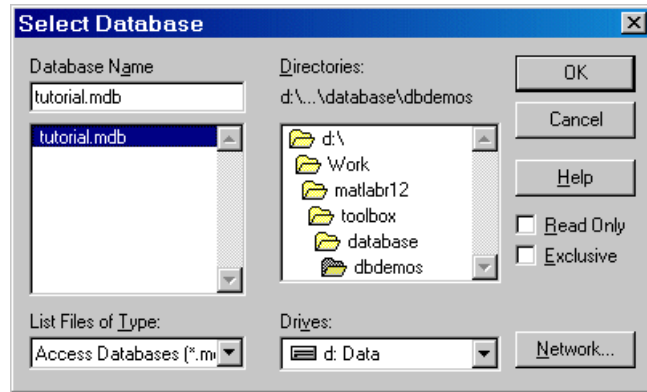


- 5 Provide a **Data Source Name** and **Description**.

For one example data source, type dbtoolboxdemo as the data source name.

Note that for some databases, the **ODBC Setup** dialog box requires you to provide additional information.

- 6 Select the database that this data source will use. Note that for some drivers, you skip this step.
  - a In the **ODBC Setup** dialog box, click **Select**.  
The **Select Database** dialog box appears.



- b Find and select the database you want to use. For the dbtoolboxdemo data source, select tutorial.mdb in \$matlabroot\toolbox\database\dbdemos.  
  
If your database resides on another system to which your PC is connected, you must first click **Network**. The **Map Network Drive** dialog box appears. Find and select the directory containing the database you want to use, and then click **Finish**. The **Map Network Drive** dialog box closes.
    - c Click **OK** to close the **Select Database** dialog box.
  - 7 In the **ODBC Setup** dialog box, click **OK**.
  - 8 Repeat steps 3 through 7 to set up the data source for the other example database, northwind.
    - In step 5, type SampleDB as the data source name.
    - In step 6, select northwind.mdb. See “SampleDB data source” in “Setting Up a Data Source for ODBC Drivers” on page 1-7 for instructions to obtain the data from Microsoft.



- 9 Click **OK** to close the **ODBC Data Source Administrator** dialog box, which now contains the `dbtoolboxdemo` and `SampleDB` data sources.

## Setting Up a Data Source for JDBC Drivers

To set up a data source for use with a Windows or UNIX system using JDBC drivers, include a pointer to the JDBC driver location in the MATLAB `$matlabroot/toolbox/local/classpath.txt` file. For example, add the following line to your `classpath.txt` file

```
/dbtools/classes111.zip
```

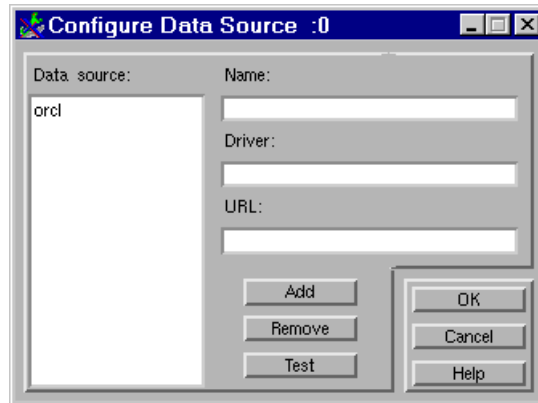
where `classes111.zip` is the file containing JDBC drivers, and it is located in the `dbtools` directory. The filename is different for each database system. The file is available from your database provider.

## Setup for Using Visual Query Builder on UNIX Platforms

If you want to use the Visual Query Builder on UNIX platforms, perform these steps to set up the JDBC data source. (You cannot use the Visual Query Builder on a Windows platform with a JDBC driver.)

- 1 Start MATLAB if it is not already running.

- 2 Access the **Configure Data Source** dialog box by typing  
confds



Any existing data sources are listed under **Data source**.

- 3 Complete the **Name**, **Driver**, and **URL** fields. For example:

**Name:** orcl

**Driver:** oracle.jdbc.driver.OracleDriver

**URL:** jdbc:oracle:thin:@144.212.123.24:1822:

- 4 Click **Add** to add the data source.
- 5 Click **Test** to establish a test connection to the data source. You are prompted to supply a username and password if the database requires it.
- 6 Click **OK** to save the changes and close the **Configure Data Source** dialog box.

To remove a data source, select it from the **Data source** list in the **Configure Data Source** dialog box, click **Remove**, and click **OK**.

## Starting the Database Toolbox

Use the Database Toolbox functions the way you would use any MATLAB function in the Command Window. For more information, see “Using Functions in the Database Toolbox” on page 3-1.

To start the Visual Query Builder, type `querybuilder`. For more information, see “Visual Query Builder” on page 2-1.



# Visual Query Builder

---

Getting Started with the Visual Query Builder (p. 2-2)	Follow the list of steps to use the Visual Query Builder (VQB). Know when to use the VQB and when to use toolbox functions.
Creating, Running, and Saving a Query (p. 2-7)	Build a query, run it, save it, set preferences including data format, use retrieved data, and clear variables.
Viewing Query Results (p. 2-16)	View results as a relational display, a chart, in a table report, and in a customized report.
Fine-Tuning Queries Using Advanced Query Options (p. 2-27)	Retrieve unique occurrences, retrieve data meeting specified criteria, order the results, use subqueries to retrieve values from multiple tables, and other options.

# Getting Started with the Visual Query Builder

The Visual Query Builder (VQB) is an easy-to-use graphical user interface for retrieving data from your database. With the VQB, you build queries to retrieve data by selecting information from lists rather than by entering MATLAB functions. The VQB retrieves the data from a database and puts it in a MATLAB cell array, structure, or numeric matrix so you can process the data using the MATLAB suite of functions. With the VQB, you can display information retrieved as cell arrays in relational tables, reports, and charts.

## Before You Start

Before using the Visual Query Builder, set up a data source. For instructions, see “Setting Up a Data Source” on page 1-7.

Note that if you use JDBC drivers on a Windows platform, you cannot use the Visual Query Builder.

## Starting the Visual Query Builder

To start the Visual Query Builder interface, type

```
querybuilder
```

at the MATLAB prompt. The Visual Query Builder opens. When you start the VQB, all fields except the **Data source** are blank. You can also start the interface using the **Start** menu or Launch Pad in the MATLAB desktop.

## Summary of Steps to Use the Visual Query Builder

To start the Visual Query Builder, type `querybuilder` at the MATLAB prompt.

\* Required step

10 View query results in table, chart, and report formats.      1\* Select data source.    2\* Select tables.    3\* Select fields to retrieve.

7 Set preferences for data retrieval.

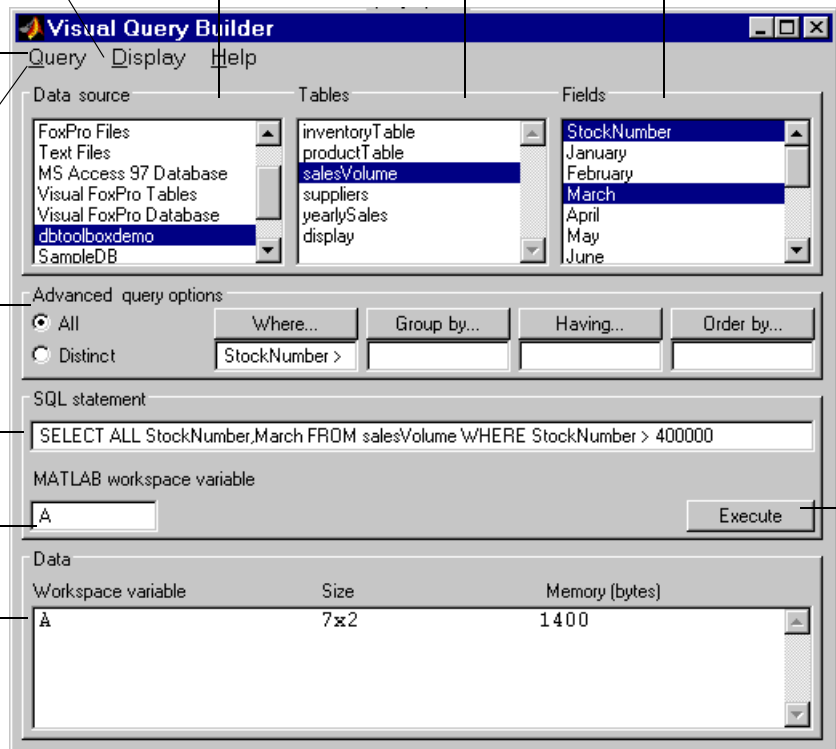
11 Save, load, and run queries.

4 Refine query.

5 View SQL statement.

6\* Assign variable for results.

9 Double-click to view query results in MATLAB Command Window.



8\* Run query.

Detailed instructions to perform these steps are in subsequent sections. Required steps are marked with an \*.

- 1\* Select the data source from which you want to import data.
- 2\* Select the tables that contain the data you want to import.
- 3\* Select the fields whose data you want to import.
- 4 Use advanced query options to refine the query.
- 5 View the SQL statement.
- 6\* Assign a MATLAB workspace variable for the results.
- 7 Set preferences for data retrieval using the **Query** menu
- 8\* Execute the query.
- 9 View the resulting data in the **MATLAB Command Window**.
- 10 View results in table, chart, and report formats using the **Display** menu.
- 11 Save and load queries using the **Query** menu.

### Quitting the Visual Query Builder

To quit using the Visual Query Builder, select **Exit** from the **Query** menu, or click the close box.

### When to Use the Visual Query Builder

If you want to retrieve information from relational databases for use in MATLAB and you are not familiar with the Structured Query Language (SQL) and database applications, use the Visual Query Builder.

If you are familiar with SQL and your database applications, use the Visual Query Builder to easily build SQL queries and import query results into MATLAB, or use Database Toolbox functions instead.

Note that when you use the VQB, it automatically generates and displays the SQL statement for the query. Therefore, you can copy and paste the SQL statements you generate using the VQB into MATLAB code that uses Database Toolbox functions.



## When to Use Database Toolbox Functions

You must use Database Toolbox functions rather than the Visual Query Builder to

- Export results from MATLAB to databases.
- Write MATLAB applications that access databases.
- Perform functions not available with the Visual Query Builder. For example, if you want to access data from tables whose names contain spaces, you need to use Database Toolbox functions since the Visual Query Builder does not support that capability.

You can also use Database Toolbox functions instead of the Visual Query Builder to import data into MATLAB.

For information on how to use the functions, see “Using Functions in the Database Toolbox” on page 3-1 and “Function Reference” on page 4-1.

## Examples for the Visual Query Builder

This document uses simple examples to demonstrate many of the Visual Query Builder features. These examples use the dbtoolboxdemo data source (tutorial database) for Microsoft Access. Instructions for setting up this data source are in “Setting Up a Data Source” on page 1-7. If you don’t have Microsoft Access, you should still be able to follow the examples because they are not complex.

If your version of Microsoft Access is different than that used in “Setting Up a Data Source” you might get different results than those presented here. If your results differ, check your version of Access and check the table and column names in your databases to see if they are the same as those used in the examples.

The examples used are

- “Creating, Running, and Saving a Query” on page 2-7
- “Viewing Query Results” on page 2-16
- “Fine-Tuning Queries Using Advanced Query Options” on page 2-27

### Running a Visual Query Builder Demo

You can run a demo of the Visual Query Builder, which illustrates its main features. In the **Visual Query Builder** dialog box, select **Demos** from the **Help** menu. Follow the instructions in the Command Window, which prompt you to press **Enter** to move through the demo.

The demo runs on Windows platforms only. It uses the dbtoolboxdemo data source (tutorial database). Instructions for setting up this data source are in “Setting Up a Data Source” on page 1-7.

If your version of Microsoft Access is different than that referred to in “Setting Up a Data Source”, you might get different results than those shown in the demo. If your results differ, check your version of Access and check the table and column names in your database to see if they are the same as those used in the demo.

### Online Help for the Visual Query Builder

While using the Visual Query Builder, get online help by

- Selecting **Visual Query Builder Help** from the **Help** menu. This documentation for the Visual Query Builder appears in the Help browser.
- Clicking **Help** in any Visual Query Builder dialog box. Detailed instructions for that dialog box appear in the Help browser.

For more information about getting help, see “Using the Help Browser” in the MATLAB documentation.

## Creating, Running, and Saving a Query

Topics covered in this section are

- “Building and Executing a Query” on page 2-7
- “Saving a Query” on page 2-10
- “Specifying Preferences for NULLS, Data Format, and Error Handling” on page 2-11
- “Using Retrieved Data in MATLAB” on page 2-15
- “Clearing Variables from the Data Area” on page 2-15

### Building and Executing a Query

Build and run a query to import data from your database into MATLAB. Then save the query for use again later.

**Before You Start.** Before using the VQB, set up a data source—see “Setting Up a Data Source” on page 1-7. The examples here use the `dbtoolboxdemo` data source.

**To Start.** To open the VQB, in the Command Window type

```
querybuilder
```

In the VQB, perform these steps to create and run a query:

- 1 From the **Data source** list box, select the data source from which you want to import data. For this example, select `dbtoolboxdemo`, which is the data source for the tutorial database.

The list includes all data sources you set up. If you do not see the data source you want to use, you need to add it—see “Setting Up a Data Source” on page 1-7.

After selecting a data source, the **Tables** in that data source appears.

- 2 From the **Tables** list box, select the table that contains the data you want to import. For this example, select salesVolume.

After selecting a table, the **Fields** (column names) in that table appear.

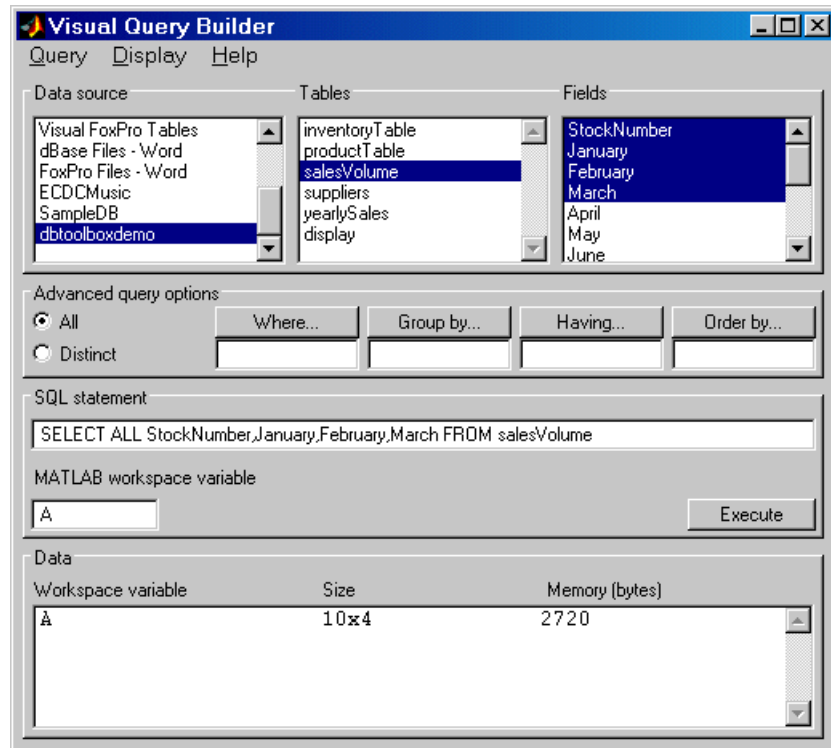
**Note:** Using the Visual Query Builder, you cannot access tables whose names include spaces. You can instead use Database Toolbox functions.

- 3 From the **Fields** list box, select the fields containing the data you want to import. To select more than one field, hold down the **Ctrl** key or **Shift** key while selecting. For this example, select the fields StockNumber, January, February, and March.

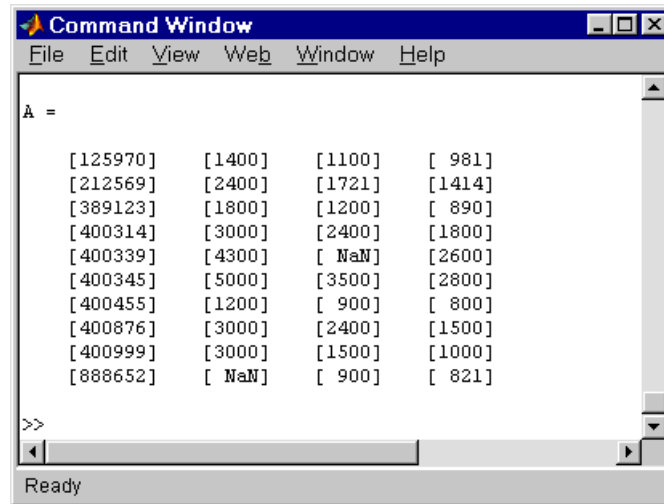
As you select items from the **Fields** list, the query appears in the **SQL statement** field.

- 4 In the **MATLAB workspace variable** field, assign a name for the data returned by the query. For this example, use A.
- 5 Click **Execute** to run the query and retrieve the data.

The query runs, retrieves data, and stores it in MATLAB, which in this example is a cell array assigned to the variable A. In the **Data** area, information about the query result appears.



- 6 Double-click A in the **Data** section. The contents of A is displayed in the **Command Window**. Another way to see the contents of A is to type A in the **Command Window**.



As an example of how to read the results, sales for item 400876 are 3000 in January, 2400 in February, and 1500 in March.

### Saving a Query

After building a query in the VQB, you can save it for later use. To save a query:

- 1 Select **Save** from the **Query** menu.

The **Save SQL Statement** dialog box appears.

- 2 Complete the **File name** field and click **Save**. For the example, type `basic` as the filename.

The query is saved with a `.qry` extension.

The MATLAB workspace variable name you assigned for the query results and the query preferences are *not* saved as part of the query.

## Using a Saved Query

To use a saved query:

- 1 Select **Load** from the **Query** menu.

The **Load SQL Statement** dialog box appears.

- 2 Select the name of the query you want to load and click **Open**. For the example, select `basic.qry`.

The VQB fields reflect the values for the saved query.

- 3 To run the query, assign a variable in the **MATLAB workspace variable** field and click **Execute**.

## Specifying Preferences for NULLS, Data Format, and Error Handling

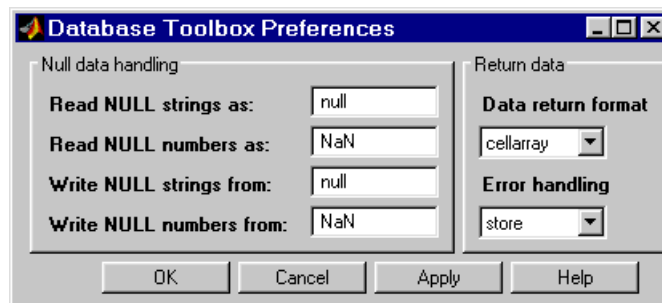
Using preferences, you can specify

- How the query builder represents NULL data
- Format of data retrieved
- Method for error notification

To set preferences:

- 1 Select **Preferences** from the **Query** menu.

The **Database Toolbox Preferences** dialog box appears, showing the current settings.



- 2 Change the current preference settings to the new values and click **OK**. For this example, make the following changes.

Preference	Description	New Value
<b>Read NULL numbers as</b>	<p>How NULL numbers in a database are represented when imported into MATLAB.</p> <p>For the new value, 0, the NULL data in the example results will appear as 0s. Previously, they appeared as NaN values.</p>	0
<b>Data return format</b>	<p>Format for data imported into MATLAB. Select a value based on the type of data you are importing, memory considerations, and your preferred method of working with retrieved data. You cannot use the <b>Display</b> menu items for the numeric and structure formats.</p> <p>Because our results are all numeric, we can change from <code>cellarray</code> to <code>numeric</code> to reduce memory required.</p>	numeric
<b>Error handling</b>	<p>Behavior for handling errors when importing data. In the Visual Query Builder, setting the value to <code>store</code> or <code>empty</code> means any errors are reported in a dialog box rather than in the <b>Command Window</b>.</p> <p>Set the value to <code>report</code>, which means that any errors from running the query will display immediately in the <b>Command Window</b>.</p>	report

For more information about these preferences, see the property descriptions on the reference page for `setdbprefs`, which is the equivalent function for setting preferences. Note that the settings for writing strings and numbers are not relevant for use in the Visual Query Builder, since you cannot export data using the Visual Query Builder. However, you can change the settings in the **Preferences** dialog box and those settings will be used if you export data using the Database Toolbox functions.



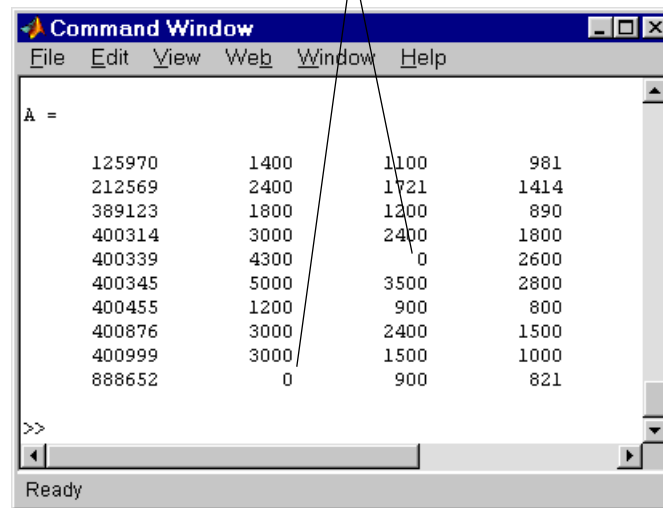
- 3 Enter a workspace variable, A, and click **Execute** to run the query again.

Information about the retrieved data appears in the **Data** section. Note that the **Memory** size of A is 320 bytes, compared to 2720 bytes when we ran the query using the previous settings for preferences. This is because we changed the **Data return format** to **numeric**, where previously it was set to **cellarray**. The **numeric** format requires far less memory than the **cellarray** format. However, the **cellarray** (or **structure**) format is required if you want to retrieve data that is not all numeric, such as strings, or if you want to use the **Display** menu items. If you use the **numeric** format to retrieve data that contains strings, the strings are returned as **NULL** values, represented by the preference you specified for **Read NULL numbers as**.

#### 4 To see the results, type A in the **Command Window**.

Results are not in brackets because data is a numeric matrix rather than a cell array.

NULL values are now represented by 0's instead of NaNs.



Note that 0s are displayed where previously there were NaNs to represent NULL values. Also note that the data is not in brackets since it is a numeric matrix rather than a cell array.

### Saving Preferences

Preferences apply to the current MATLAB session. They are not saved with a query. It is a good practice to verify the preference settings before you run a query, especially if it retrieves a large amount of data.

Another way to set preferences is by using the `setdbprefs` function. To use the same preferences whenever you run MATLAB, include the `setdbprefs` function in your startup file.

## Using Retrieved Data in MATLAB

When you execute a query, MATLAB retrieves the data and stores it in the variable name you provided. Using preferences, you specify the data return format as `cellarray`, `structure`, or `numeric`. Cell arrays and structures support mixed data types, but require more memory and are processed more slowly than numeric matrices. Use the numeric format if the data you are retrieving consists only of numeric data or if the nonnumeric data is not relevant. With the numeric format, any strings are converted to the representation specified in the `NullNumberRead` preference, for example, `NaN`.

For more information, see “Working with Cell Arrays in MATLAB” on page 3-36.

## Clearing Variables from the Data Area

Variables in the **Data** area include those you assigned for query results, as well as any variables you assigned in the **Command Window**. The variables do not appear in the **Data** area until you execute a query. They remain in the **Data** area until you clear them in the **Command Window** using the `clear` function, and then execute a query.

## Viewing Query Results

After running a query in the Visual Query Builder, you can view the retrieved data in the **MATLAB Command Window**, as described in step 7 of “Creating, Running, and Saving a Query” on page 2-7. In addition, the **VQB Display** menu provides additional options for viewing data:

- “Relational Display of Data” on page 2-16
- “Chart Display of Results” on page 2-20; for example, a pie chart
- “Report Display of Results in a Table” on page 2-23
- “Customized Display of Results in the Report Generator” on page 2-25

To use these **Display** menu items, the data return format preference must be set to `cellarray`.

To use the saved query from the earlier example, `basic.qry`, with the **Display** menu examples in this section, change the preference, and load and run the query:

- 1** Select **Query -> Preferences**.
- 2** In the **Preferences** dialog box, change the **Data return format** to `cellarray` and click **OK**.
- 3** Select **Query -> Load**.
- 4** In the **Load SQL Statement** dialog box, select the **File name**, `basic.qry`, and click **Open**.
- 5** In the VQB, type a value for the **MATLAB workspace variable**, for example, `A`, and then click **Execute**.

### Relational Display of Data

- 1** After executing a query, select **Data** from the **Display** menu.

The query results appear in a figure window.

Figure No. 2

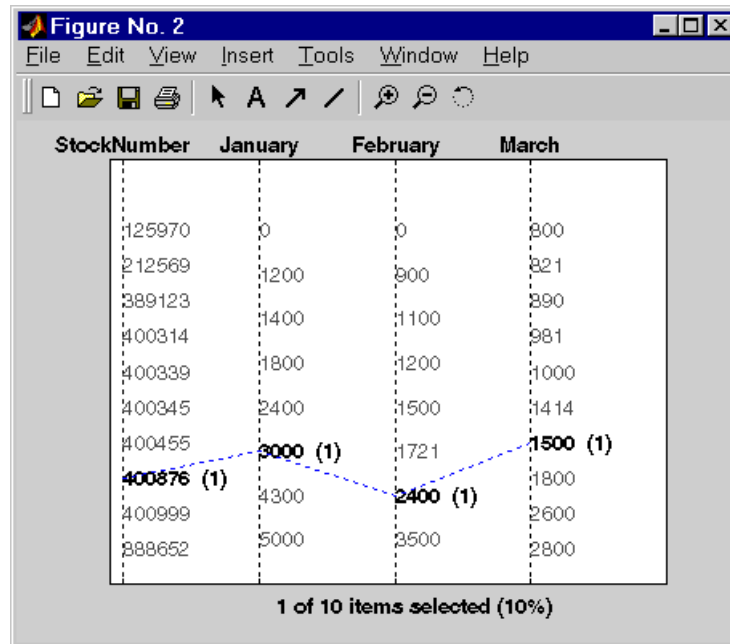
StockNumber	January	February	March
125970	0	0	800
212569	1200	900	821
389123	1400	1100	890
400314	1800	1200	981
400339	1800	1200	1000
400345	2400	1500	1414
400455	3000	1721	1500
400876	4300	2400	1800
400999	5000	3500	2600
888652	5000	3500	2800

Click on a text object

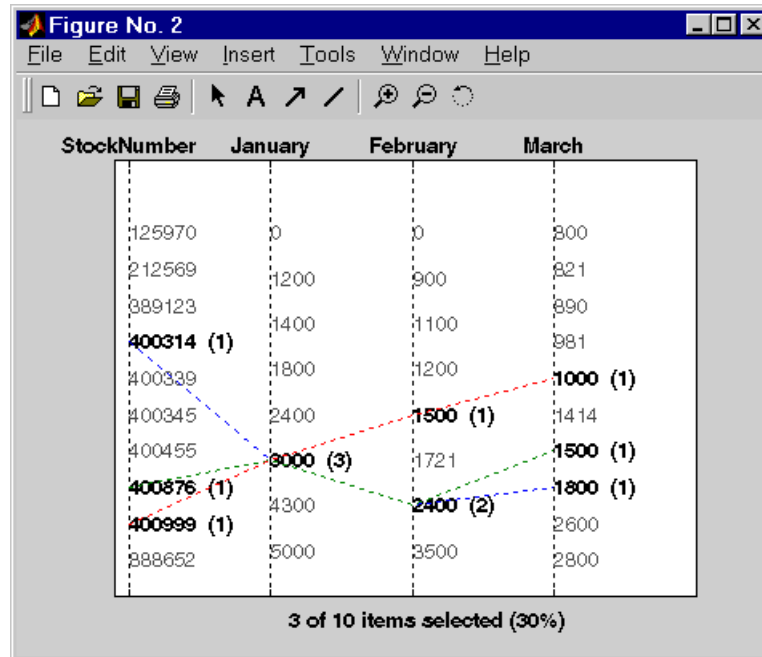
The display shows only the *unique* values for each field, so you do *not* read each row as a single record. For the `basic.qry` example, there are 10 entries in the `StockNumber` field, 8 entries in the `January` and `February` fields, and 10 entries in the `March` field, corresponding to the number of unique values in those fields.

- 2 Click a value in the display, for example `StockNumber 400876`, to see the associated values.

The data associated with the selected value is shown in bold and connected via a dotted line. For example, sales for item 400876 are 3000 in January, 2400 in February, and 1500 in March.



As another example, click 3000 in the January field. It shows three different items with sales of 3000 units in January: 400314, 400876, and 400999.

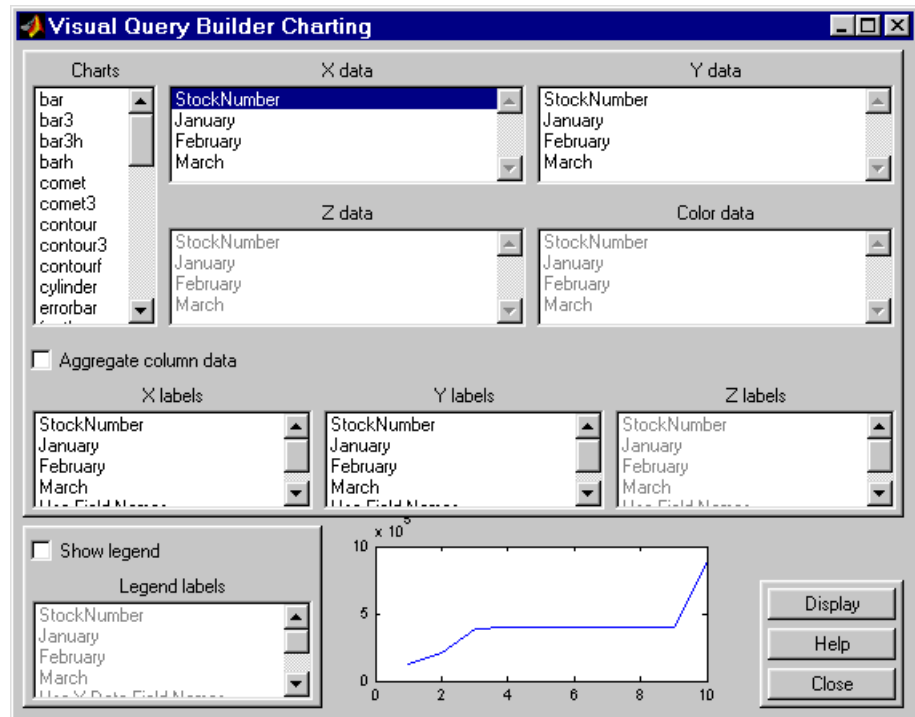


- 3 Because the display is presented in a MATLAB figure window, you can use some MATLAB figure features. For example, you can print the figure and annotate it. To print it, select **File -> Print**. You can also use **File -> Page Setup** and **File -> Print Preview**. For more information, use the **Figure** window **Help** menu.
- 4 If the query results include many entries, the display might not effectively show all of them. You can stretch the window to make it larger, modify the query so there are fewer results, or display the results in a table (see “Report Display of Results in a Table” on page 2-23).

## Chart Display of Results

- 1 After executing a query, select **Chart** from the **Display** menu.

The **Charting** dialog box appears.



- 2 Select the type of chart you want to display from the **Charts** listbox (plot is the default). For example, select pie to display a pie chart.

The preview of the chart at the bottom of the dialog box shows the result of your selection. For this example, the pie chart replaces the plot line, with each stock item appearing in a different color.



- 3 Select the data you want to display in the chart from the **X data**, **Y data**, and **Z data** listboxes. For the pie chart example, select March from the **X data** list box to display a pie chart of March data.

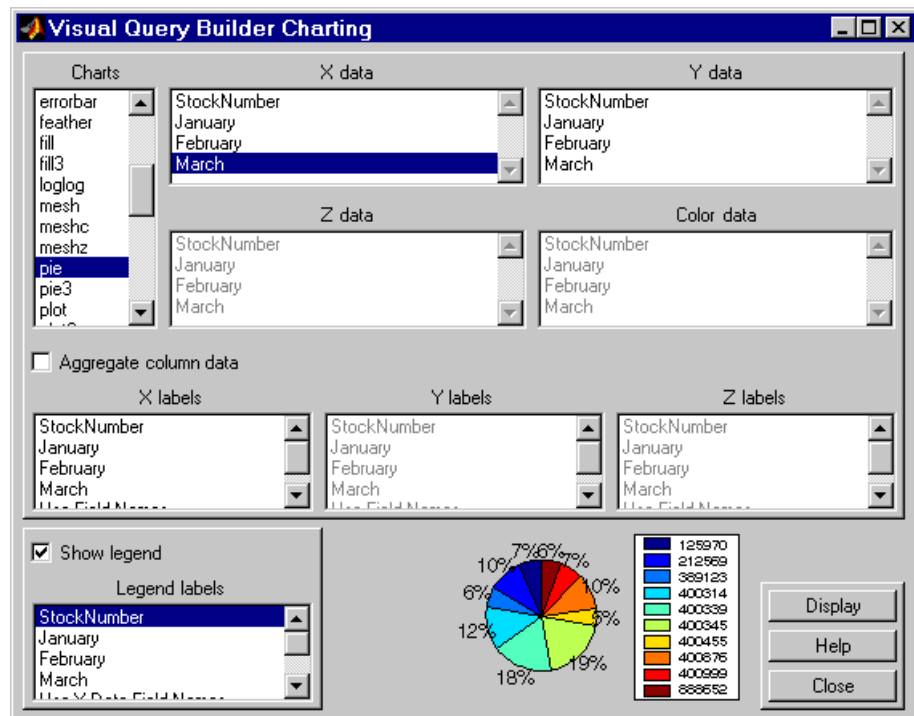
The preview of the chart at the bottom of the dialog box reflects the selection you made. For this example, the pie chart shows percentages for March data.

- 4 To display a legend, which maps the colors to the stock numbers, check the **Show legend** check box.

The **Legend labels** become available for you to select from.

- 5 Select StockNumber from the **Legend labels** listbox.


A legend appears in the preview of the chart. You can drag and move the legend in the preview

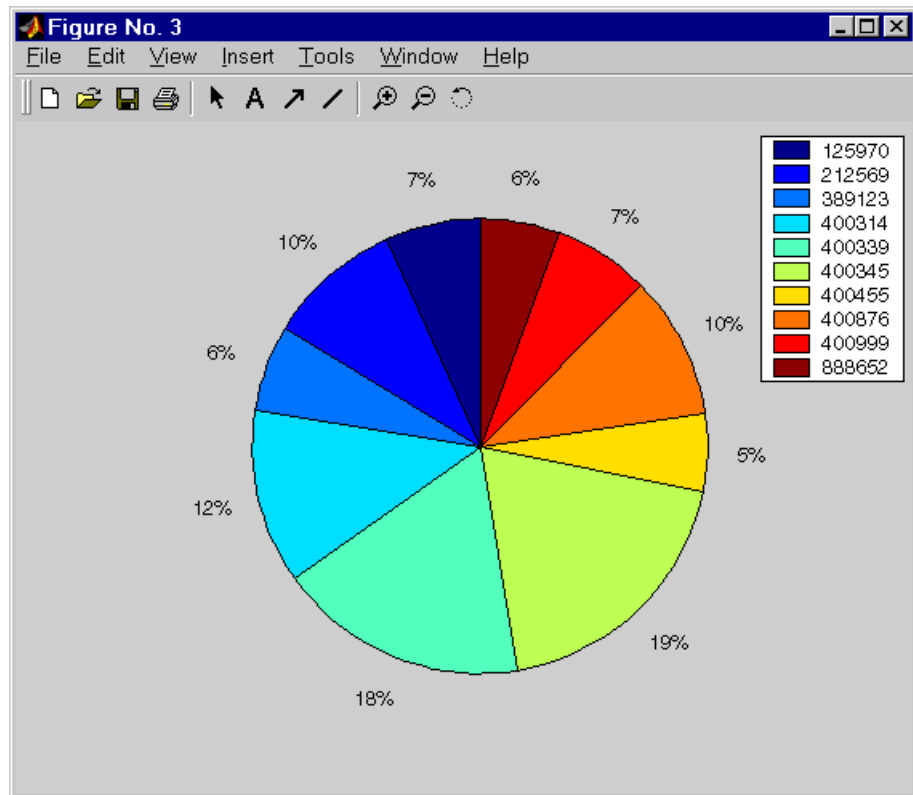


### 6 Click **Display**.

The pie chart appears in a figure window. Because the display is presented in a MATLAB figure window, you can use some MATLAB figure features such as printing or annotating the figure. To print the figure, select **File -> Print**. You can also use **File -> Page Setup** and **File -> Print Preview**. For more information, use the **Figure** window's **Help** menu.

For example:

- Resize the window by dragging any corner or edge.
- Drag the legend to another position.
- Annotate the chart using the **Tools** menu and the annotation buttons in the toolbar . For more information, use the **Figure** window's **Help** menu.



- 7 Click **Close** to close the **Charting** dialog box.

There are many different ways to present the query results using the chart feature. For more information, click **Help** in the **Charting** dialog box.

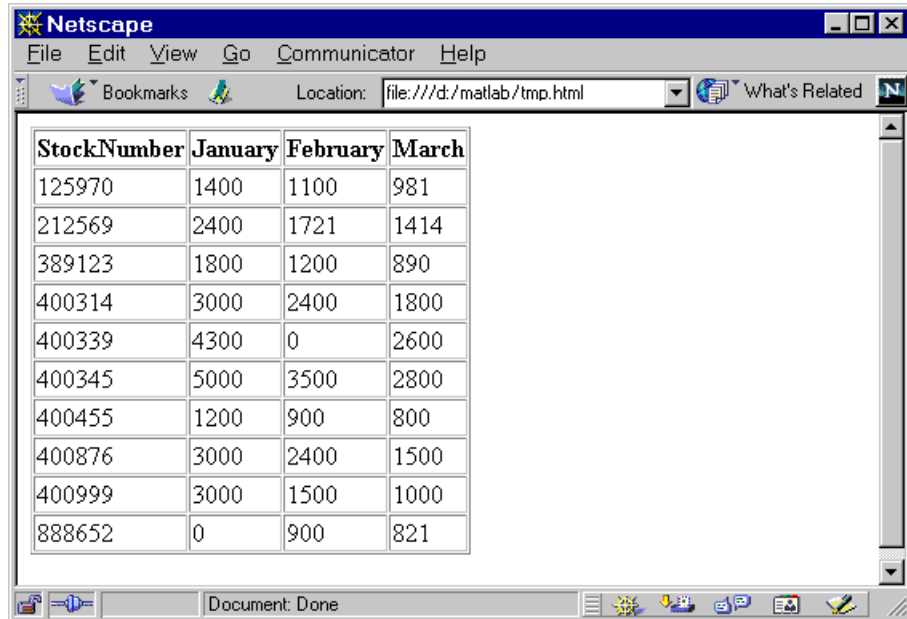
## Report Display of Results in a Table

The report display presents the results in your system's default Web browser.

- 1 Because some browser configurations do not launch automatically, start your Web browser before using this feature.

- 2 After executing a query, select **Report** from the **Display** menu.

The query results appear as a table. If you have the Report Generator product installed, the appearance of the report will be slightly different than what is shown here.



The screenshot shows a Netscape browser window with a table of stock sales data. The table has four columns: StockNumber, January, February, and March. The data is as follows:

StockNumber	January	February	March
125970	1400	1100	981
212569	2400	1721	1414
389123	1800	1200	890
400314	3000	2400	1800
400339	4300	0	2600
400345	5000	3500	2800
400455	1200	900	800
400876	3000	2400	1500
400999	3000	1500	1000
888652	0	900	821

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.

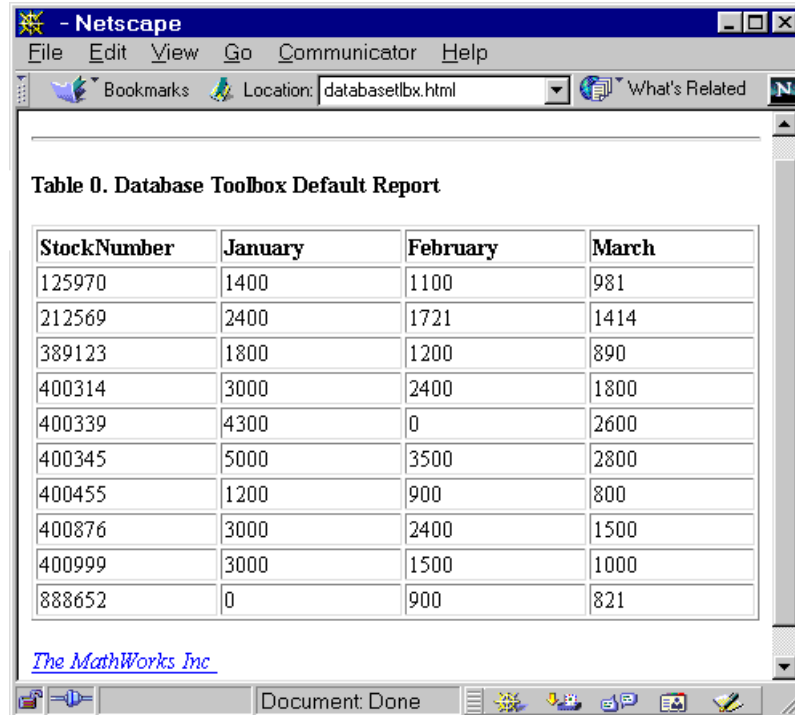
- 3 Use your Web browser to save the report as an HTML page if you want to view it later. If you do not save it, the report will be overwritten the next time you select **Report** from the **Display** menu. To print the report, use the print features in your Web browser.

## Customized Display of Results in the Report Generator

- 1 Because some browser configurations do not launch automatically, start your Web browser before using this feature.
- 2 After executing a query, select **Report Generator** from the **Display** menu.  
The **Setup File List** dialog box appears.
- 3 Select `$matlabroot\toolbox\database\vpq\databaset1bx.rpt` from the list.
- 4 To modify the report format, click **Edit**. Click the **Help** button in the dialog box for more information about this and other features of the Report Generator.

- 5 To view the report, click **Report**.

The report appears in your system's default Web browser.



The screenshot shows a Netscape browser window with the title bar '- Netscape'. The address bar shows 'Location: databasetlbx.html'. The main content area displays a table with the following data:

StockNumber	January	February	March
125970	1400	1100	981
212569	2400	1721	1414
389123	1800	1200	890
400314	3000	2400	1800
400339	4300	0	2600
400345	5000	3500	2800
400455	1200	900	800
400876	3000	2400	1500
400999	3000	1500	1000
888652	0	900	821

Below the table, there is a link: [The MathWorks Inc](#). The browser's status bar at the bottom shows 'Document: Done' and various icons.

This example shows a report of sales volume over 3 months by product stock number. From the report, you can see that sales for item 400876 are 3000 in January, 2400 in February, and 1500 in March.

## Fine-Tuning Queries Using Advanced Query Options

Use advanced query options in the Visual Query Builder for

- “Retrieving Unique Occurrences” on page 2-27.
- “Retrieving Information That Meets Specified Criteria” on page 2-29.
- “Presenting Results in Specified Order” on page 2-38.
- “Creating Subqueries for Values from Multiple Tables” on page 2-42.
- “Creating Queries for Results from Multiple Tables” on page 2-47.
- “Other Features in Advanced Query Options” on page 2-51.

For more information about advanced query options, select **Help** in any of the dialog boxes for the options.

### Retrieving Unique Occurrences

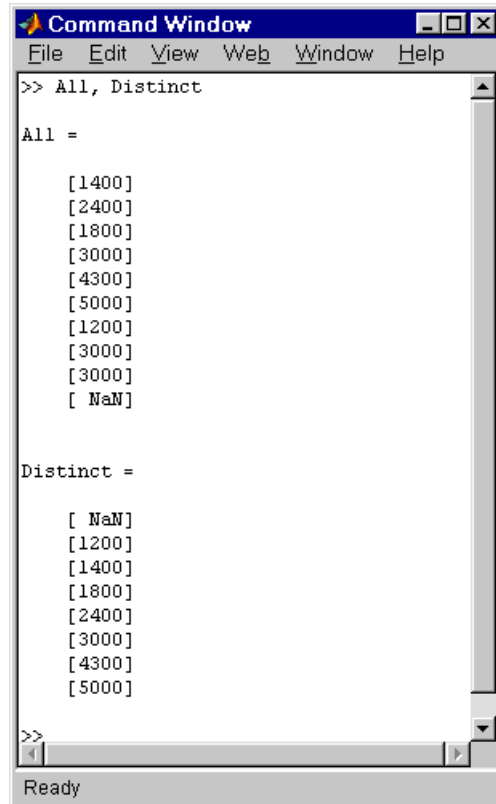
In the Visual Query Builder **Advanced query options**, select **Distinct** to limit results to only unique occurrences. Select **All** to retrieve all occurrences. For example:

- 1** Set **Preferences**; for this example, set **Data return format** to cellarray and **Read NULL numbers as** to NaN.
- 2** Select the **Data source**; for this example, dbtoolboxdemo.
- 3** Select the **Tables**; for this example, SalesVolume.
- 4** Select the **Fields**; for this example, January.
- 5** Run the query to retrieve all occurrences.
  - a** In **Advanced query options**, select **All**.
  - b** Assign a **MATLAB workspace variable**; for this example, All.
  - c** Click **Execute**.

- 6 Run the query to retrieve only unique occurrences.
  - a In **Advanced query options**, select **Distinct**.
  - b Assign a **MATLAB workspace variable**, for this example, **Distinct**.
  - c Click **Execute**.
- 7 In the **Data** area, the **Workspace variable - Size** shows 10x1 for All and 8x1 for **Distinct**.



- 8 In the **Command Window**, type `All, Distinct` to display the query results.



```
>> All, Distinct

All =

    [1400]
    [2400]
    [1800]
    [3000]
    [4300]
    [5000]
    [1200]
    [3000]
    [3000]
    [ NaN]

Distinct =

    [ NaN]
    [1200]
    [1400]
    [1800]
    [2400]
    [3000]
    [4300]
    [5000]

>>
Ready
```

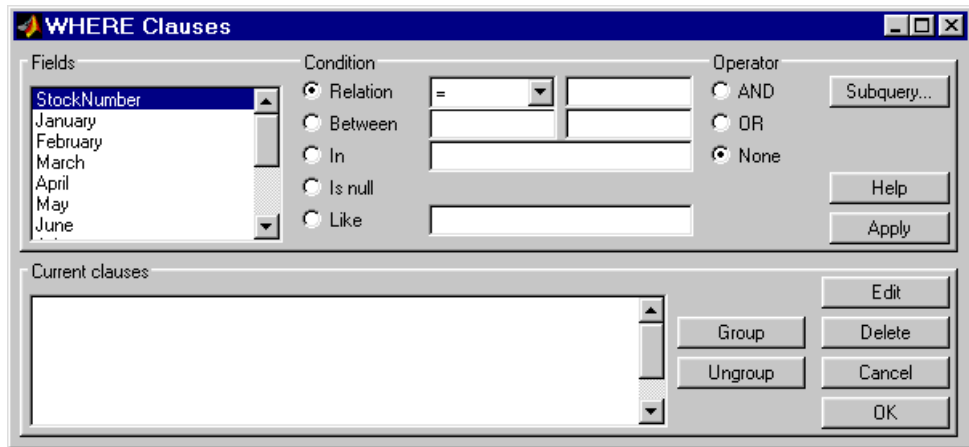
The value 3000, appears three times in `All`, but appears only once in `Distinct`.

## Retrieving Information That Meets Specified Criteria

Use the **Where** field in **Advanced query options** to retrieve only the information that meets the criteria you specify. This example uses `basic.qry`, that was created and saved as explained in “Creating, Running, and Saving a Query” on page 2-7. It limits the results to those stock numbers greater than 400000 and less than 500000:

- 1 Load `basic qry`. For instructions, see “Using a Saved Query” on page 2-11.
- 2 Set **Preferences**; for this example, set **Data return format** to `cellarray` and **Read NULL numbers as** to `NaN`.
- 3 In **Advanced query options**, click **Where**.

The **Where Clauses** dialog box appears.

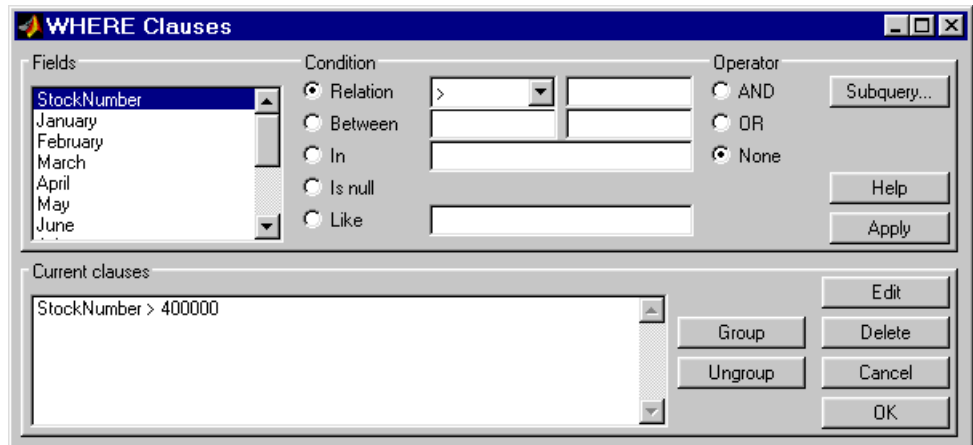


- 4 Select the **Fields** whose values you want to restrict. For example, select `StockNumber`.

- 5 Use **Condition** to specify the criteria. For example, specify that the StockNumber be greater than 400000.
  - a Select **Relation**.
  - b From the drop-down list to the right of **Relation**, select >.
  - c In the field to the right of the drop-down list, type 400000.

The screenshot shows the 'WHERE Clauses' dialog box. The 'Fields' list on the left includes 'StockNumber', 'January', 'February', 'March', 'April', 'May', and 'June'. The 'Condition' section has 'Relation' selected, with a dropdown menu showing '>' and a text box containing '400000'. The 'Operator' section has 'None' selected. The 'Current clauses' area is empty. Buttons for 'Apply', 'Help', 'Subquery...', 'Edit', 'Group', 'Ungroup', 'Delete', 'Cancel', and 'OK' are visible.

- d Click **Apply**.  
The clause appears in the **Current clauses** area.



- 6 You can add another condition. First you edit the current clause to add the AND operator to it, and then you provide the new condition.
  - a Select StockNumber > 400000 from **Current clauses**.
  - b Click **Edit** (or double-click the StockNumber entry in **Current clauses**). The **Condition** reflects the StockNumber clause.
  - c For **Operator**, select **AND**.
  - d Click **Apply**.  
The **Current clauses** updates to show  
StockNumber > 400000 AND
- 7 Add the new condition. For example, specify that StockNumber must also be less than 500000.
  - a From **Fields**, select StockNumber.
  - b Select **Relation** from **Condition**.
  - c From the drop-down list to the right of **Relation**, select <.
  - d In the field to the right of the drop-down list, type 500000.

**e Click Apply.**

The **Current clauses** area now shows

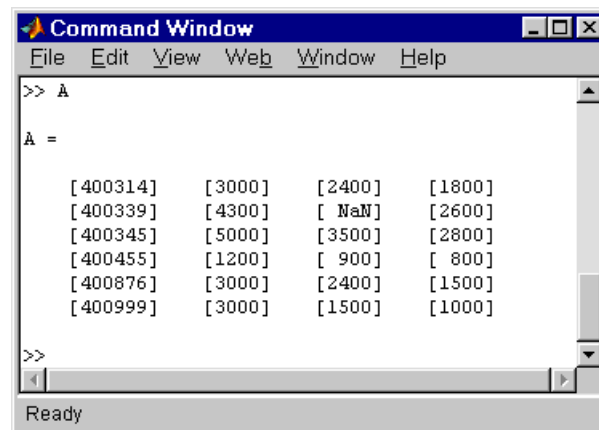
```
StockNumber > 400000 AND
StockNumber < 500000
```

**8 Click OK.**

The **Where Clauses** dialog box closes. The **Where** field and the **SQL statement** in the **Visual Query Builder** dialog box reflect the where clause you specified.

**9 Assign a MATLAB workspace variable; for example, A.****10 Click Execute.**

The results are a 6-by-4 matrix.

**11 To view the results, double click the A in the Data area of the VQB and they appear in the Command Window. Compare these to the results for all stock numbers, which is a 10-by-4 matrix (see step 6 in “Building and Executing a Query”).**


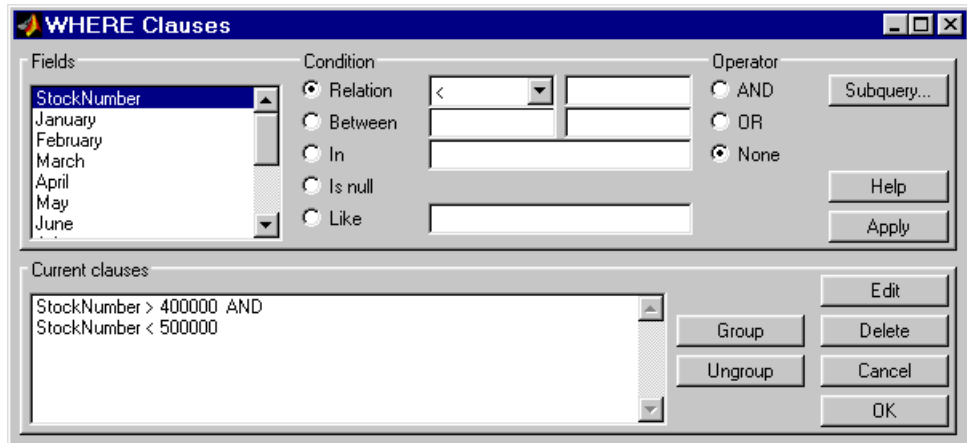
```
Command Window
File Edit View Web Window Help
>> A
A =
    [400314]    [3000]    [2400]    [1800]
    [400339]    [4300]    [ NaN]    [2600]
    [400345]    [5000]    [3500]    [2800]
    [400455]    [1200]    [ 900]    [ 800]
    [400876]    [3000]    [2400]    [1500]
    [400999]    [3000]    [1500]    [1000]
>>
Ready
```

**12 Select Save from the Query menu and name this query basic\_where.qry for use with subsequent examples.**

### Grouping Criteria

In the **Where Clauses** dialog box, you can group together constraints so that the group of constraints is evaluated as a whole in the query. For the example, `basic_where.qry`, where `StockNumber` is greater than 400000 and less than 500000, modify the query to group constraints. The new query will retrieve results where sales in any of the 3 months is greater than 1500 units, as long as sales for each of the 3 months is greater than 1000 units.

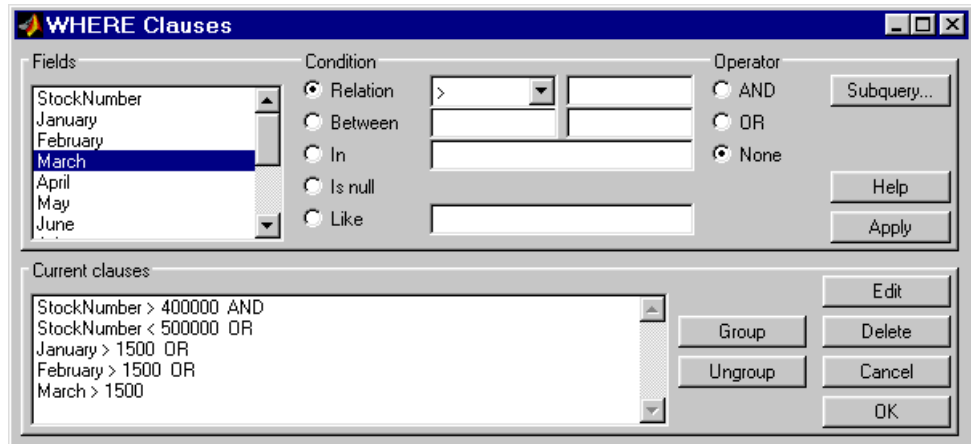
Click **Where** in the Visual Query Builder. The **Where Clauses** dialog box appears as follows, to retrieve data where the `StockNumber` is greater than 400000 and less than 500000.



- 1 Add the criteria that retrieves data where sales in any of the 3 months is greater than 1500 units.
  - a In **Current clauses**, select `StockNumber < 500000`, and then click **Edit**.
  - b For **Operator**, select **OR**, and then click **Apply**.
  - c In **Fields**, select **January**. For **Relation**, select **>** and type 1500 in the field for it. For **Operator**, select **OR**, and then click **Apply**.
  - d In **Fields**, select **February**. For **Relation**, select **>** and type 1500 in the field for it. For **Operator**, select **OR**, and then click **Apply**.

- e In **Fields**, select March. For **Relation**, select > and type 1500 in the field for it. Then click **Apply**.

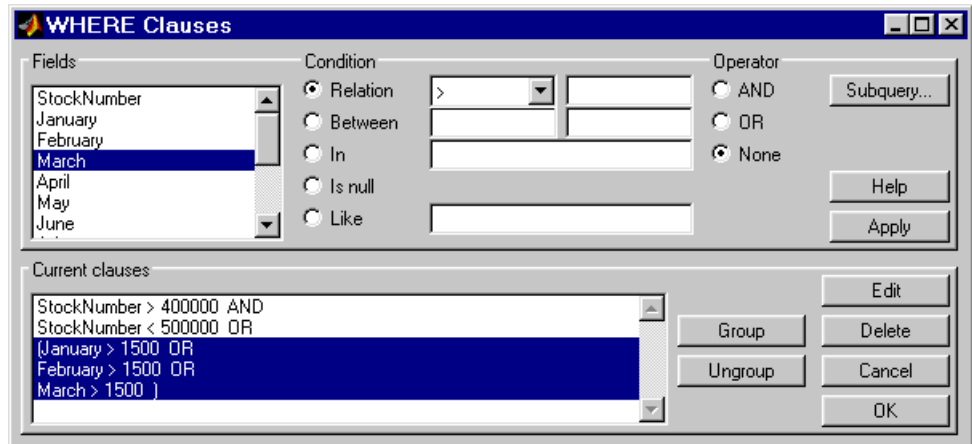
The **Where Clauses** dialog box appears as follows.



- 2 Group the criteria requiring any of the months to be greater than 1500 units.

- a In **Current clauses**, select the statement January >1500 OR.
- b **Shift**+click to also select February > 1500 OR.
- c **Shift**+click to also select March > 1500.
- d Click **Group**.

An opening parenthesis is added before January, and a closing parenthesis is added after March > 1500, signifying that these statements are evaluated as a whole.

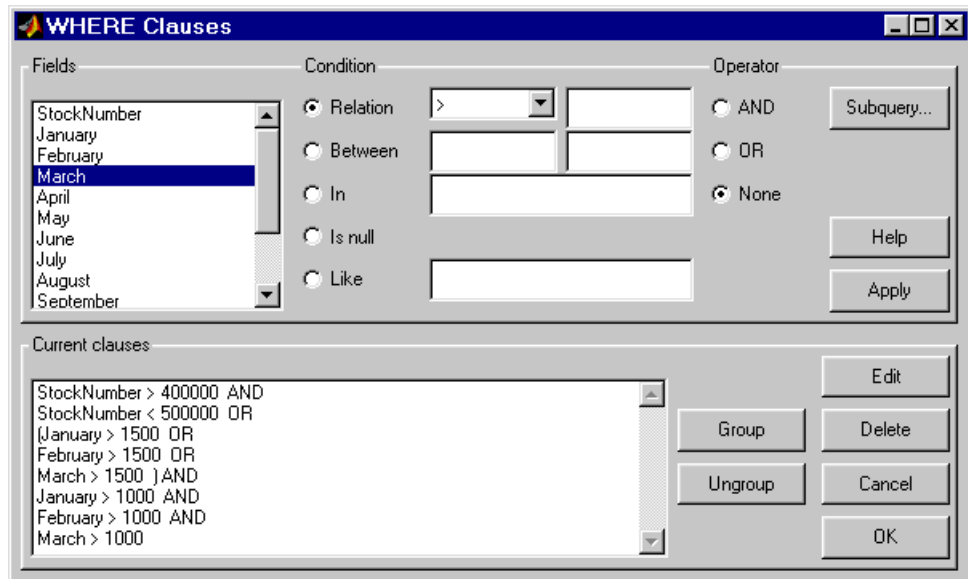


- 3 Add the criteria that retrieves data where sales in each of the 3 months is greater than 1000 units.
  - a In **Current clauses**, select the statement March >1500), and then click **Edit**.
  - b For **Operator**, select AND, and then click **Apply**.
  - c In **Fields**, select January. For **Relation**, select > and type 1000 in the field for it. For **Operator**, select AND, and then click **Apply**.
  - d In **Fields**, select February. For **Relation**, select > and type 1000 in the field for it. For **Operator**, select AND, and then click **Apply**.



- e In **Fields**, select March. For **Relation**, select > and type 1000 in the field for it. Then click **Apply**.

The **Where clauses** dialog box appears as follows.



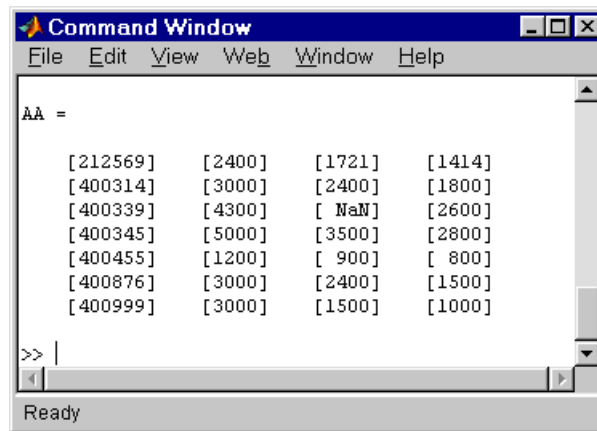
- f Click **OK**.

The **Where Clauses** dialog box closes. The **SQL statement** in the **Visual Query Builder** dialog box reflects the modified where clause. Because the clause is so long, you have to use the right arrow key in the field to see all of the contents.

- 4 Assign a **MATLAB workspace variable**, for example, AA.
- 5 Click **Execute**.

The results are a 7-by-4 matrix.

- 6 To view the results, type AA in the **Command Window**.



**Removing Grouping.** To remove grouping criteria in the **Where Clauses** dialog box, in **Current clauses**, select all of the statements in the group, and then click **Ungroup**. The parentheses are removed from the statements.

For the above example, to remove the grouping, select

```
(January > 1000 AND
```

and then **Shift**+click to also select

```
February > 1000 AND  
March > 1000)
```

Then click **Ungroup**. The three statements are no longer grouped.

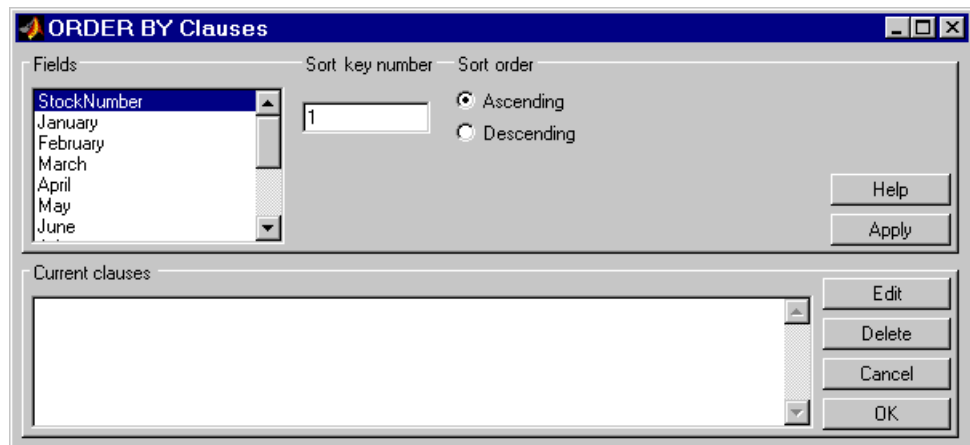
## Presenting Results in Specified Order

By default, the order of the rows in the query results depends on their order in the database, which is effectively random. Use the **Order by** field in **Advanced query options** to specify the order in which results appear. This example uses `basic_where.qry` that was created and saved in the example presented in “Retrieving Information That Meets Specified Criteria” on page 2-29.

This example sorts the results of `basic_where.qry`, so that January is the primary sort field, February the secondary, and March the last. Results for January and February are ascending, and results for March are descending:

- 1 Load `basic_where.qry`. For instructions, see “Using a Saved Query” on page 2-11.
- 2 Set **Preferences**. For this example, set **Data return format** to `cellarray` and **Read NULL numbers** as to `NaN`.
- 3 In **Advanced query options**, click **Order by**.

The **Order By Clauses** dialog box appears.



- 4 For the **Fields** whose results you want to specify the order of, specify the **Sort key number** and **Sort order**. For example, specify January as the primary sort field, with results displayed in ascending order.
  - a From **Fields**, select January.
  - b For **Sort key number**, type 1.
  - c For **Sort order**, select **Ascending**.
  - d Click **Apply**.

The **Current clauses** area now shows  
January ASC

- 5 Specify February as the second sort field, with results displayed in ascending order.
  - a From **Fields**, select February.
  - b For **Sort key number**, type 2.
  - c For **Sort order**, select **Ascending**.
  - d Click **Apply**.

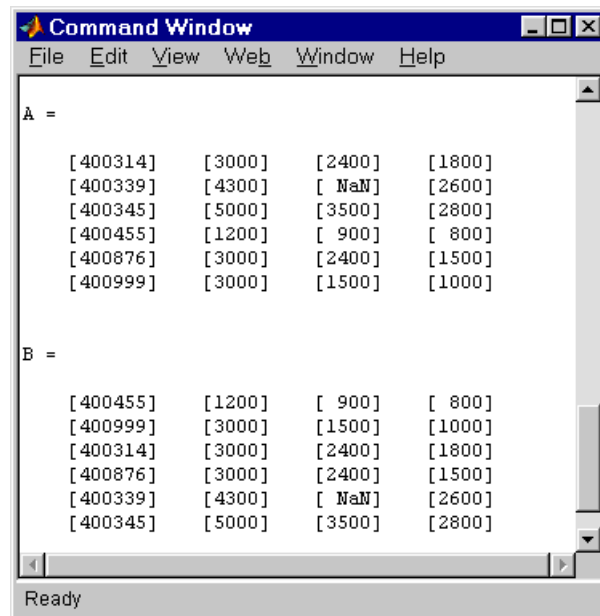
The **Current clauses** area now shows

```
January ASC  
February ASC
```
- 6 Specify March as the third sort field, with results displayed in descending order.
  - a From **Fields**, select March.
  - b For **Sort key number**, type 3.
  - c For **Sort order**, select **Descending**.
  - d Click **Apply**.

The **Current clauses** area now shows

```
January ASC  
February ASC  
March DESC
```
- 7 Click **OK**.

The **Order By Clauses** dialog box closes. The **Order by** field and the **SQL statement** in the **Visual Query Builder** reflect the order by clause you specified.
- 8 Assign a **MATLAB workspace variable**, for example, B.
- 9 Click **Execute**.
- 10 To view the results, type B in the **Command Window**. Compare these to the unordered query results, shown as A.



```
Command Window
File Edit View Web Window Help

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]

Ready
```

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 and the highest value, 5000 (for item number for 400345) appears last.

For items 400999, 400314, and 400876, January sales were equal at 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.

## Creating Subqueries for Values from Multiple Tables

Use the **Where** feature in **Advanced query options** to specify a subquery, which further limits a query by using values found in other tables. This is referred to as nested SQL. With the VQB, you can only include one subquery; use Database Toolbox functions to use multiple subqueries.

This example uses `basic.qry` (see “Creating, Running, and Saving a Query” on page 2-7). It retrieves sales volumes for the product whose description is Building Blocks. The table used for `basic.qry`, `salesVolume`, has sales volumes and a stock number field, but not a product description field. Another table, `productTable`, has the product description and stock number, but not the sales volumes. Therefore, the query needs to look at `productTable` to get the stock number for the product whose description is Building Blocks, and then has to look at the `salesVolume` table to get the sales volume values for that stock number:

- 1 Load `basic.qry`. For instructions, see “Using a Saved Query” on page 2-11.

This creates a query that retrieves the values for January, February, and March sales for all stock numbers.

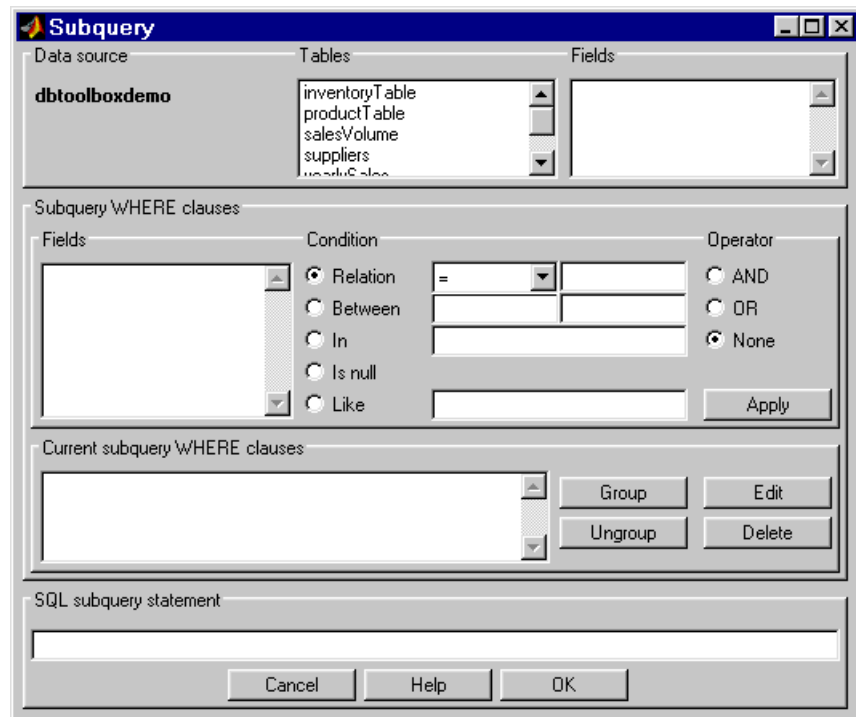
- 2 Set **Preferences**. For this example, set **Data return format** to `cellarray` and **Read NULL numbers as** to `NaN`.

- 3 In **Advanced query options**, click **Where**.

The **Where Clauses** dialog box appears.

- 4 Click **Subquery**.

The **Subquery** dialog box appears.



- 5 From **Tables**, select the table that contains the values you want to associate. In this example, select `productTable`, which contains the association between the stock number and the product description.

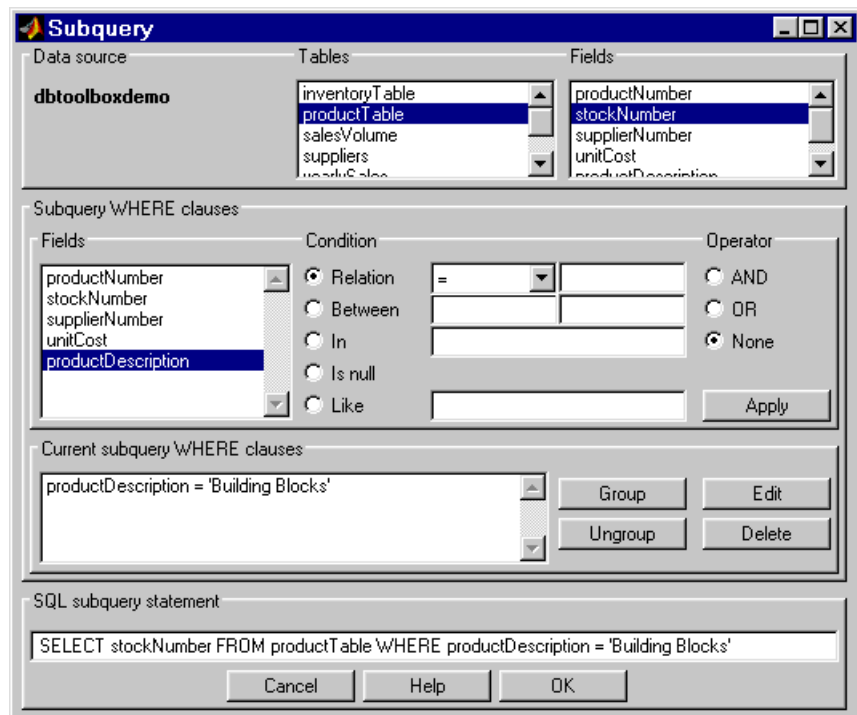
The fields in that table appear.

- 6 From **Fields**, select the field that is common to this table and the table from which you are retrieving results (the table you selected in the **Visual Query Builder** dialog box). In this example, select `stockNumber`.

This begins creating the **SQL subquery statement** to retrieve the stock number from `productTable`.

- 7** Create the condition that limits the query. In this example, limit the query to those product descriptions that are Building Blocks.
  - a** In **Subquery Where clauses**, select productDescription from **Fields**.
  - b** For **Condition**, select **Relation**.
  - c** From the drop-down list to the right of **Relation**, select =.
  - d** In the field to the right of the drop-down list, type 'Building Blocks' (include the single quotation marks to denote it is a string).
  - e** Click **Apply**.

The clause appears in the **Current subquery Where clauses** area and updates the **SQL subquery statement**.



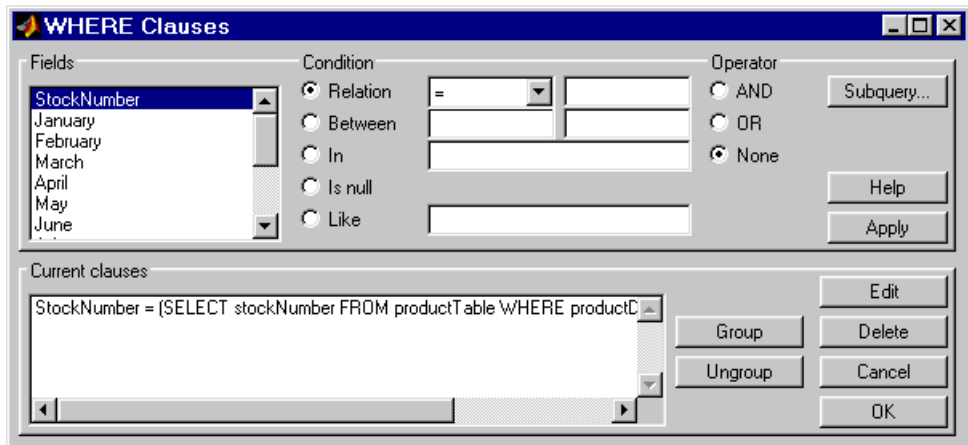


- 8** In the **Subquery** dialog box, click **OK**.

The **Subquery** dialog box closes.

- 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**.

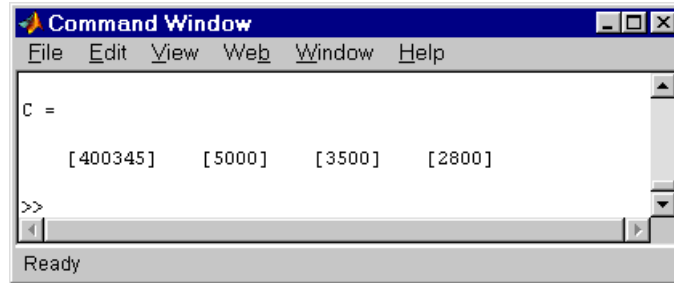
This closes the **Where Clauses** dialog box and updates the **SQL statement** in the **Visual Query Builder** dialog box.

- 11** In the **Visual Query Builder** dialog box, assign a **MATLAB workspace variable**, for example, C.

- 12** Click **Execute**.

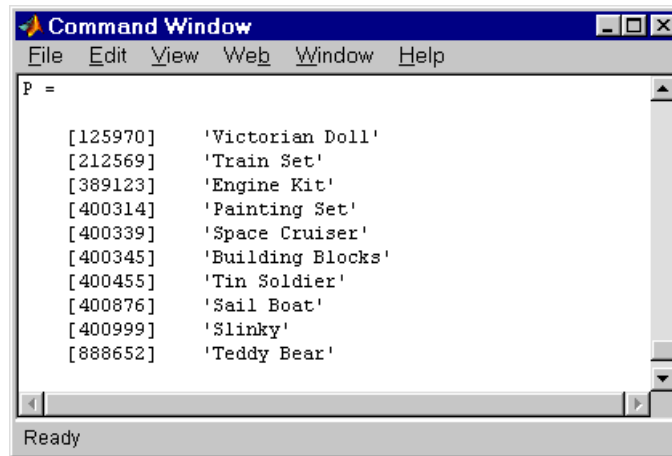
The results are a 1-by-4 matrix.

- 13** Type C at the prompt in the **MATLAB Command Window** to see the results.



- 14** The results are for item 400345, which has the product description Building Blocks, although that is not evident from the results. To verify that the product description is actually Building Blocks, run this simple query.
- a** Select dbtoolboxdemo as the **Data source**. This clears the VQB selections made during a previous query.
  - b** Select productTable from **Tables**.
  - c** Select stockNumber and productDescription from **Fields**.
  - d** Assign a **MATLAB workspace variable**, for example, P.

- e Click **Execute**.
- f Type P at the prompt in the **Command Window** to view the results.



The results show that item 400345 has the product description Building Blocks. The following section, “Creating Queries for Results from Multiple Tables” creates a query that includes the product description in the results.

## Creating Queries for Results from Multiple Tables

You can select multiple tables to create a query whose results include values from both tables. This is called a *join* operation in SQL.

This example retrieves sales volumes by product description. The example is very similar to the example in “Creating Subqueries for Values from Multiple Tables” on page 2-42. The difference is that this example creates a query that uses both tables in order to include the product description rather than the stock number in the results.

The table `salesVolume`, has sales volumes and a stock number field, but not a product description field. Another table, `productTable`, has the product description and the stock number, but not sales volumes. Therefore, the query needs to retrieve data from both tables and equate the stock number from `productTable` with the stock number from the `salesVolume` table:

**1** Set **Preferences**. For this example, set **Data return format** to cellarray and **Read NULL numbers as** to NaN.

**2** Select the **Data source**, for this example, dbtoolboxdemo. This clears the VQB selections made during a previous query

The tables in that data source appear in **Tables**.

**3** From **Tables**, select the tables from which you want to retrieve data. For example, **Ctrl**+click on productTable and salesVolume to select both tables.

The fields (columns) in those tables appear in **Fields**. Note that the field names now include the table names. For example, productTable.stockNumber is the field name for the stock number in the product table, and salesVolume.StockNumber is the field name for the stock number in the sales volume table.

**4** From **Fields**, select these fields to be included in the results. For example, **Ctrl**+click on productTable.productDescription, salesVolume.January, salesVolume.February, and salesVolume.March.

**5** In **Advanced query options**, click **Where** to make the necessary associations between fields in different tables. For example, the where clause equates the productTable.stockNumber with the salesVolume.StockNumber so that the product description is associated with sales volumes in the results.

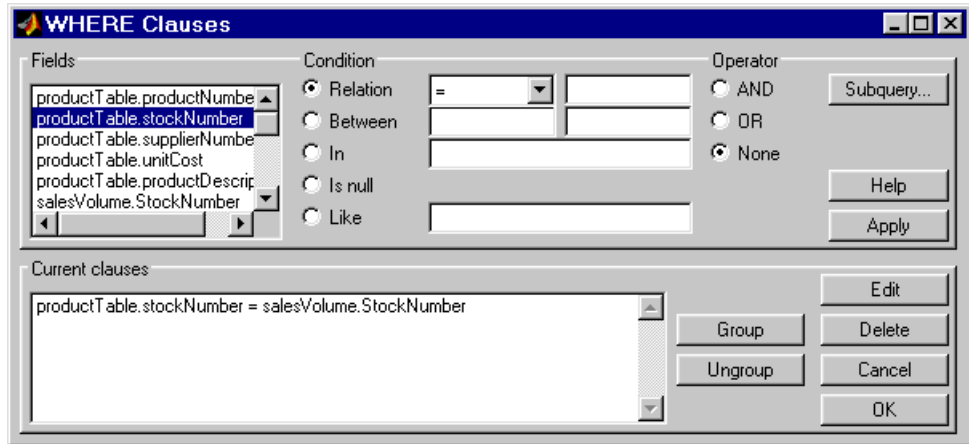
The **Where Clauses** dialog box appears.

**6** In the **Where Clauses** dialog box.

- a** Select productTable.stockNumber from **Fields**.
- b** For **Condition**, select **Relation**.
- c** From the drop-down list to the right of **Relation**, select =.
- d** In the field to the right of the drop-down list, type salesVolume.StockNumber.

- e Click **Apply**.

The clause appears in the **Current clauses** area.



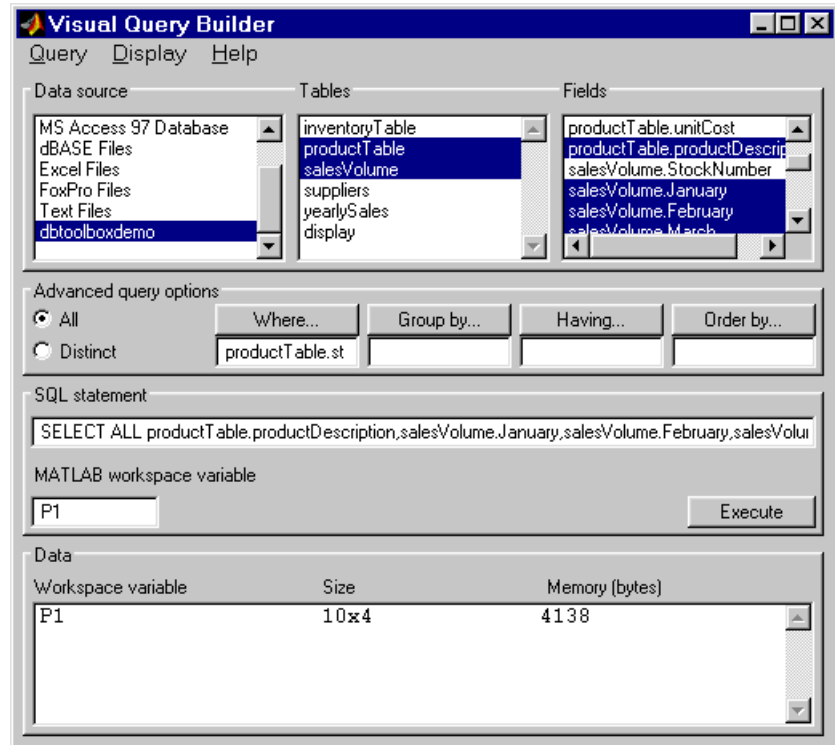
- f Click **OK**.

The **Where Clauses** dialog box closes. The **Where** field and **SQL statement** in the **Visual Query Builder** dialog box reflect the where clause.

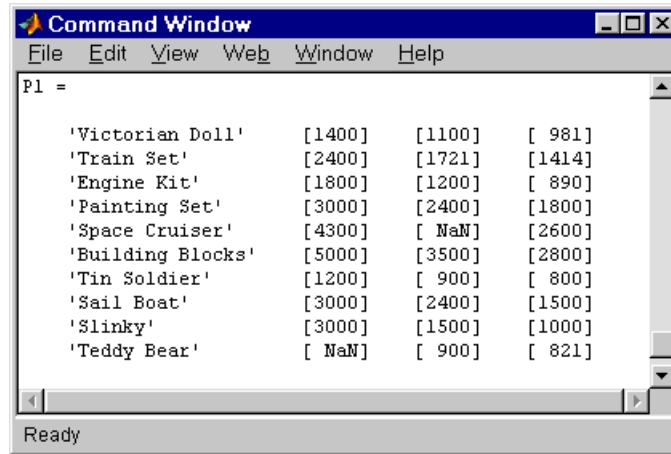
- 7 Assign a **MATLAB workspace variable**, for example, P1.

8 Click **Execute** to run the query.

The results are a 10-by-4 matrix.



- 9 Type P1 at the prompt in the **Command Window** to see the results.



## Other Features in Advanced Query Options

For more information about advanced query options, select the option and then click **Help** in the resulting dialog box. For example, click **Group by** in **Advanced query options**, and then click **Help** in the **Group by Clauses** dialog box.





# Using Functions in the Database Toolbox

---

When first using the toolbox, follow the simple examples in this section consecutively. Once you are familiar with the process, go directly to the example of interest. To run these examples, you need to set up the specified data source—for instructions, see “Setting Up a Data Source” on page 1-7. If your version of Microsoft Access is different than the one used here, you might get different results. M-files containing functions used in some of these examples are in `matlab\toolbox\database\dbdemos`. As you work with the examples, you can open the M-files to see the functions and copy them, or you can run the M-files to see the results. For more information on the functions used in these examples, type `doc` followed by the function name, or see “Function Reference” on page 4-1.

Importing Data into MATLAB from a Database (p. 3-2)	Import data from the <code>SampleDB</code> data source, including setting the format for retrieved data.
Viewing Information About the Imported Data (p. 3-8)	View information retrieved from the <code>SampleDB</code> data source, such as number of rows and column names.
Exporting Data from MATLAB to a New Record in a Database (p. 3-11)	Export a new record from MATLAB and commit it to the <code>SampleDB</code> data source.
Replacing Existing Data in a Database MATLAB (p. 3-17)	Update an existing record in the <code>SampleDB</code> data source.
Exporting Multiple New Records from MATLAB (p. 3-19)	After importing data from the <code>dbtoolboxdemo</code> data source, export multiple records to a different table.
Accessing Metadata (p. 3-23)	Get information about the <code>dbtoolboxdemo</code> data source.
Performing Driver Functions (p. 3-30)	Create driver objects and set and get the properties (does not require you to set up a data source).
About Objects and Methods for the Database Toolbox (p. 3-33)	Use object-oriented methods with the Database Toolbox.
Working with Cell Arrays in MATLAB (p. 3-36)	Simple examples for the toolbox, if you are unfamiliar with MATLAB cell arrays, used for mixed data types.

## Importing Data into MATLAB from a Database

In this example, you connect to and import data from a database. Specifically, you connect to the SampleDB data source, and then import country data from the customers table in the northwind sample database. You learn to use these Database Toolbox functions:

- database
- exec
- fetch
- logintimeout
- ping
- setdbprefs

If you want to see or copy the functions for this example, or if you want to run the set of functions, use the M-file

matlab\toolbox\database\dbdemos\dbimportdemo.m:

- 1 If you did not already do so, set up the data source SampleDB according to the directions in “Setting Up a Data Source” on page 1-7.
- 2 In MATLAB, set the maximum time, in seconds, you want to allow the MATLAB session to try to connect to a database. This prevents the MATLAB session from hanging up if a database connection fails.

Enter the function *before* you connect to a database.

Type

```
logintimeout(5)
```

to specify the maximum allowable connection time as 5 seconds. If you are using a JDBC connection, the function syntax is different. For more information, see `logintimeout`.

MATLAB returns

```
ans=  
    5
```

When you use the `database` function in the next step to connect to the database, MATLAB tries to make the connection. If it cannot connect in 5 seconds, it stops trying.

**3** Connect to the database by typing

```
conn = database('SampleDB', '', '')
```

In this example, you define a MATLAB variable, `conn`, to be the returned connection object. This connection stays open until you close it with the `close` function.

For the `database` function, you provide the name of the database, which is the data source `SampleDB` for this example. The other two arguments for the `database` function are `username` and `password`. For this example, they are empty strings because the `SampleDB` database does not require a `username` or `password`.

If you are using a JDBC connection, the `database` function syntax is different. For more information, see the [database reference page](#).

For a valid connection, MATLAB returns information about the connection object.

```
conn =  
    Instance: 'SampleDB'  
    UserName: ''  
    Driver: []  
    URL: []  
    Constructor: [1x1  
                 com.mathworks.toolbox.database.databaseConnect]  
    Message: []  
    Handle: [1x1 sun.jdbc.odbc.JdbcOdbcConnection]  
    Timeout: 5  
    AutoCommit: 'on'  
    Type: 'Database Object'
```

#### 4 Check the connection status by typing

```
ping(conn)
```

MATLAB returns status information about the connection, indicating that the connection was successful.

```
DatabaseProductName: 'ACCESS'  
DatabaseProductVersion: '04.00.0000'  
JDBCDriverName: 'JDBC-ODBC Bridge (odbcjt32.dll)'  
JDBCDriverVersion: '2.0001 (04.00.6019)'  
MaxDatabaseConnections: 64  
CurrentUserName: 'admin'  
DatabaseURL: 'jdbc:odbc:SampleDB'  
AutoCommitTransactions: 'True'
```

#### 5 Open a cursor and execute an SQL statement by typing

```
curs = exec(conn, 'select country from customers')
```

In the `exec` function, `conn` is the name of the connection object. The second argument, `select country from customers`, is a valid SQL statement that selects the country column of data from the customers table.

The `exec` function returns a cursor object. In this example, you assign the returned cursor object to the MATLAB variable `curs`.

```

curs =
    Attributes: []
           Data: 0
 DatabaseObject: [1x1 database]
           RowLimit: 0
           SQLQuery: 'select country from customers'
           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

```

The data in the cursor object is stored in MATLAB.

#### 6 Specify the format of retrieved data by typing

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

In this example, the returned data contains strings so the data format must support strings, which `cellarray` does. If the returned data contains only numerics or if the nonnumeric data is not relevant, you could instead specify the numeric format, which uses less memory.

#### 7 Import data into MATLAB by typing

```
curs = fetch(curs, 10)
```

`fetch` is the function that imports data. It has the following two arguments in this example:

- `curs`, the cursor object returned by `exec`.
- `10`, the maximum number of rows you want to be returned by `fetch`. The `RowLimit` argument is optional. If `RowLimit` is omitted, MATLAB imports all remaining rows. When importing large quantities of data, rather than importing all the rows at once, import the data using multiple fetches and include the `rowlimit` argument to improve speed and memory usage.

In this example, `fetch` reassigns the cursor object containing the rows of data returned by `fetch` to the variable `curs`. MATLAB returns information about the cursor object.

```
curs =  
    Attributes: []  
        Data: {10x1 cell}  
DatabaseObject: [1x1 database]  
    RowLimit: 0  
    SQLQuery: 'select country from customers'  
    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: com.mathworks.toolbox.database.fetchTheData]
```

The `curs` object contains an element, `Data`, that in turn contains the rows of data in the cell array. You can tell that `Data` contains 10 rows and 1 column.

- 8** Display the `Data` element in the cursor object, `curs`. Assign the data element, `curs.Data` to the variable `AA`. Type

```
AA = curs.Data
```

MATLAB returns

```
AA =  
    'Germany'  
    'Mexico'  
    'Mexico'  
    'UK'  
    'Sweden'  
    'Germany'  
    'France'  
    'Spain'  
    'France'  
    'Canada'
```

Now you can use MATLAB to perform operations on the returned data. For more information, see “Working with Cell Arrays in MATLAB” on page 3-36.

- 9 At this point, you can go to the next example. If you want to stop working now and resume with the next example at a later time, close the cursor and the connection. Type

```
close(curs)
```

```
close(conn)
```

## Viewing Information About the Imported Data

In this example, you view information about the data you imported and close the connection. You learn to use these Database Toolbox functions:

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

If you want to see or copy the functions for this example, or if you want to run the set of functions, use the M-file

`matlab\toolbox\database\dbdemos\dbinfodemo.m`:

- 1** If you are continuing directly from the previous example (“Importing Data into MATLAB from a Database” on page 3-2), skip this step. Otherwise, if the cursor and connection are not open, type the following to continue with this example.

```
conn = database('SampleDB', '', '');  
curs = exec(conn, 'select country from customers');  
setdbprefs('DataReturnFormat', 'cellarray');  
curs = fetch(curs, 10);
```

- 2** View the number of rows in the data set you imported by typing

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

MATLAB returns

```
numrows =  
    10
```

`rows` returns the number of rows in the data set, which is 10 in this example.



- 3 View the number of columns in the data set by typing

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

MATLAB returns

```
numcols =  
    1
```

`cols` returns the number of columns in the data set, which is one in this example.

- 4 View the column names for the columns in the data set by typing

```
colnames = columnnames(curs)
```

MATLAB returns

```
colnames =  
    'country'
```

`columnnames` returns the names of the columns in the data set. In this example, there is only one column, and therefore only one column name, 'country', is returned.

- 5 View the width of the column (size of field) in the data set by typing

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

MATLAB returns

```
colsize =  
    15
```

`width` returns the column width for the column number you specify. Here, the width of column 1 is 15.

- 6** You can use a single function to view multiple attributes for a column by typing

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

MATLAB returns

```
attributes =  
    fieldName: 'country'  
    typeName: 'VARCHAR'  
    typeValue: 12  
    columnWidth: 15  
    precision: []  
    scale: []  
    currency: 'false'  
    readOnly: 'false'  
    nullable: 'true'  
    Message: []
```

Note that if you had imported multiple columns, you could include a `colnum` argument to specify the number of the column for which you want the information.

- 7** Close the cursor by typing

```
close(curs)
```

Always close a cursor when you are finished with it to avoid using memory unnecessarily and to ensure there are enough available cursors for other users.

- 8** At this point, you can go to the next example. If you want to stop working now and resume with the next example at a later time, close the connection. Type

```
close(conn)
```

## Exporting Data from MATLAB to a New Record in a Database

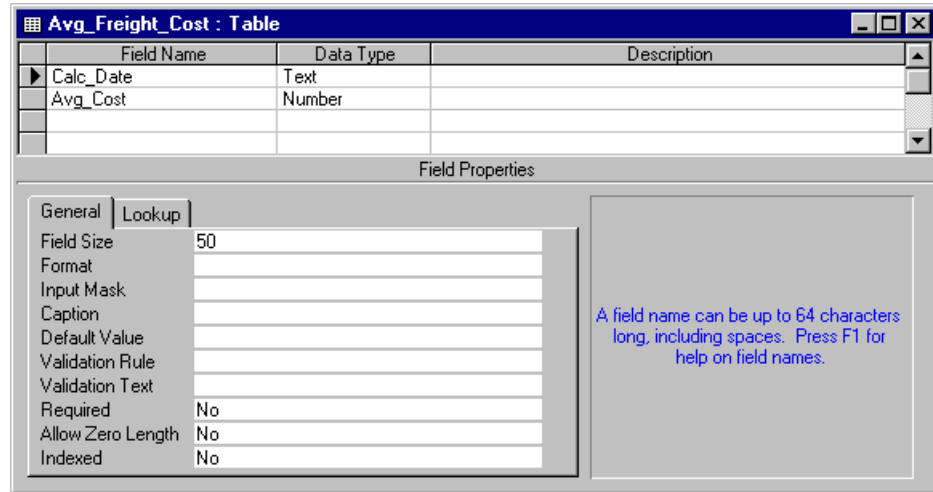
In this example, you retrieve a set of data, perform a simple calculation on the data using MATLAB, and export the results as a new record to another table in the database. Specifically, you retrieve freight costs from an orders table, calculate the average freight cost, put the data into a cell array to export it. Then export the data (the average freight cost and the date the calculation was made) to an empty table.

You learn to use these Database Toolbox functions:

- `get`
- `insert`
- `setdbprefs`

- 1 Create a table in Microsoft Access into which you will export MATLAB results.
  - a Check the properties of the northwind database to be sure it is writable, that is, *not* read only.
  - b Open the northwind database in Microsoft Access.
  - c Create a new table that has two columns, `Calc_Date` and `Avg_Cost`.

- d For the Calc\_Date field, use the default **Data Type**, which is Text, and for the Avg\_Cost field, set the **Data Type** to Number.



- e Save the table as Avg\_Freight\_Cost and close it. Access warns you that there is no primary key, but you do not need one. If you do designate a primary key, you can only run the example once because Access prevents you from inserting the same record twice.

If you need more information about how to create a table in Access, see Microsoft Access help or written documentation. Also, refer to the “Note” on page 1-3 for information about the use of spaces in table and column names and the use of reserved words.

After creating the table in Access, if you want to run a set of functions similar to this example, use the M-file  
matlab\toolbox\database\dbdemos\dbinsertdemo.m.

- 2 If you are continuing from the previous example “Viewing Information About the Imported Data” on page 3-8, skip this step. Otherwise, connect to the data source, SampleDB. Type  

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

- 3** In MATLAB, set the format for retrieved data to numeric by typing
- ```
setdbprefs('DataReturnFormat','numeric')
```

In this example, the returned data will contain only a column of numbers so the data format can be numeric, which is needed to perform calculations on the data.

- 4** Import the data on which you want to perform calculations. Specifically, import the freight column of data from the orders table. To keep the example simple, import only three rows of data. Type

```
curs = exec(conn, 'select freight from orders');  
curs = fetch(curs, 3);
```

- 5** View the data you imported by typing

```
AA = curs.Data
```

MATLAB returns

```
AA =  
    32.3800  
    11.6100  
    65.8300
```

- 6** Calculate the average freight cost. First, assign the number of rows in the array to the variable numrows. Then calculate the average, assigning the result to the variable meanA. Type

```
numrows = rows(curs);  
meanA = sum(AA(:))/numrows
```

MATLAB returns

```
meanA =  
    36.6067
```

- 7** Assign the date on which this calculation was made to the variable D by typing

```
D = '20-Jan-2002';
```

- 8** Assign the date and mean to a cell array, which you will export to the database. A cell array is required because the date information is a string.

Unlike importing data, you do not specify the export format using `setdbprefs`, but instead use standard MATLAB operations to define it. Put the date in the first cell by typing

```
exdata(1,1) = {D}
```

MATLAB returns

```
exdata =  
    '20-Jan-2002'
```

Put the mean in the second cell by typing

```
exdata(1,2) = {meanA}
```

MATLAB returns

```
exdata =  
    '20-Jan-2002'    [36.6067]
```

- 9 Define the names of the columns to which you will be exporting data. In this example, the column names are those in the `Avg_Freight_Cost` table you created earlier, `Calc_Date` and `Avg_Cost`. Assign the cell array containing the column names to the variable `colnames`. Type

```
colnames = {'Calc_Date', 'Avg_Cost'};
```

- 10 Before you export data from MATLAB, determine the current status of the `AutoCommit` flag for the database. The status of the `AutoCommit` flag determines if the database data will be automatically committed or not. If the flag is off, you can undo an update.

Verify the status of the `AutoCommit` flag using the `get` function by typing

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

MATLAB returns

```
ans =  
    on
```

The `AutoCommit` flag is set to on so exported data will be automatically committed. In this example, keep the `AutoCommit` flag on; for a Microsoft Access database, this is the only option.

- 11** Export the data into the Avg\_Freight\_Cost table. For this example, type `insert(conn, 'Avg_Freight_Cost', colnames, exdata)`

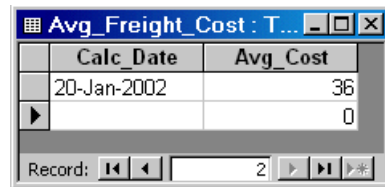
where `conn` is the connection object for the database to which you are exporting data. In this example, `conn` is `SampleDB`, which is already open. However, if you export to a different database that is not open, use the database function to connect to it before exporting the data.

`Avg_Freight_Cost` is the name of the table to which you are exporting data. In the `insert` function, you also include the `colnames` cell array and the cell array containing the data you are exporting, `exdata`, both of which you defined in the previous steps. Note that you do not define the type of data you are exporting; the data is exported in its current MATLAB format.

Running `insert` appends the data as a new record at the end of the `Avg_Freight_Cost` table.

If you get an error, it may be because the table is open in design mode in Access. Close the table in Access and repeat the `insert` function.

- 12** In Microsoft Access, view the Avg\_Freight\_Cost table to verify the results.



|  | Calc_Date   | Avg_Cost |
|--|-------------|----------|
|  | 20-Jan-2002 | 36       |
|  |             | 0        |

Record: 2

Note that the `Avg_Cost` value was rounded to a whole number to match the properties of that field in Access.

- 13** Close the cursor by typing `close(curs)`

Always close a cursor when you are finished with it to avoid using memory unnecessarily and to ensure there are enough available cursors for other users.

**14** At this point, you can go to the next example. If you want to stop working now and resume with the next example at a later time, close the connection.

Type

```
close(conn)
```

Do not delete or change the Avg\_Freight\_Cost table in Access because you will use it in the next example.



## Replacing Existing Data in a Database MATLAB

In this example, you update existing data in the database with exported data from MATLAB. Specifically, you update the date you previously imported into the Avg\_Freight\_Cost table.

You learn to use these Database Toolbox functions:

- close
- update

If you want to see or copy the functions for this example, or if you want to run a similar set of functions, use the M-file

matlab\toolbox\database\dbdemos\dbupdatedemo.m:

- 1 If you are continuing directly from the previous example (“Exporting Data from MATLAB to a New Record in a Database” on page 3-11), skip this step. Otherwise, type

```
conn = database('SampleDB', '', '');
colnames = {'Calc_Date', 'Avg_Cost'};
D = '20-Jan-2002';
meanA = 36.6067;
exdata = {D, meanA}
```

MATLAB returns

```
exdata =
    '20-Jan-2002'    [36.6067]
```

- 2 Assume that the date in the Avg\_Freight\_Cost table is incorrect and instead should be 19-Jan-2002. Type

```
D = '19-Jan-2002'
```

- 3 Assign the new date value to the cell array, newdata, which contains the data you will export. Type

```
newdata(1,1) = {D}
```

MATLAB returns

```
newdata =
    '19-Jan-2002'
```

- 4 Identify the record to be updated in the database. To do so, define an SQL where statement and assign it to the variable `whereclause`. The record to be updated is the record that has 20-Jan-2002 for the `Calc_Date`.

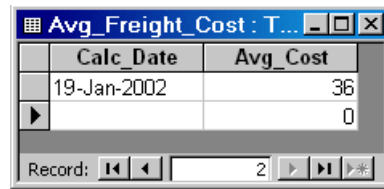
```
whereclause = 'where Calc_Date = ''20-Jan-2002'''
```

Because the date string is within a string, two single quotation marks surround the date instead of just single quotation mark. MATLAB returns

```
whereclause =  
    where Calc_Date = '20-Jan-2002'
```

- 5 Export the data, replacing the record whose `Calc_Date` is 20-Jan-2002.  

```
update(conn, 'Avg_Freight_Cost', colnames, newdata, whereclause)
```
- 6 In Microsoft Access, view the `Avg_Freight_Cost` table to verify the results.



| Calc_Date   | Avg_Cost |
|-------------|----------|
| 19-Jan-2002 | 36       |
|             | 0        |

- 7 Close the cursor and disconnect from the database.

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

Always close a connection when you are finished with it to avoid using memory unnecessarily and to ensure there are enough available connections for other users.

## Exporting Multiple New Records from MATLAB

In this example, you import multiple records, manipulate the data in MATLAB, and then export it to a different table in the database. Specifically, you import sales figures for all products, by month, into MATLAB. Then you compute the total sales for each month. Finally, you export the monthly totals to a new table.

You learn to use these Database Toolbox functions:

- `insert`
- `setdbprefs`

If you want to see or copy the functions for this example, or if you want to run a similar set of functions, use the M-file

```
matlab\toolbox\database\dbdemos\dbinsert2demo.m:
```

- 1 If you did not already do so, set up the data source `dbtoolboxdemo` according to the directions in “Setting Up a Data Source” on page 1-7. This data source uses the tutorial database.
- 2 Check the properties of the tutorial database to be sure it is writable, that is, *not* read only.
- 3 Connect to the database by typing

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

You define the returned connection object as `conn`. You do not need a username or password to access the `dbtoolboxdemo` database.

- 4 Specify preferences for the retrieved data by using the `setdbprefs` function. Set the data return format to `numeric` and specify that any NULL value read from the database is to be converted to a 0 in MATLAB.

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

Note that when you specify `DataReturnFormat` as `numeric`, the value for `NullNumberRead` must also be numeric, such as 0, and cannot be a string, such as `NaN`.

- 5** Import the sales figures. Specifically, import all data from the salesVolume table. Type

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

- 6** To get a sense of the data you imported, view the column names in the fetched data set. Type

```
columnnames(curs)
```

MATLAB returns

```
ans =  
    'Stock Number', 'January', 'February', 'March', 'April',  
    'May', 'June', 'July', 'August', 'September', 'October',  
    'November', 'December'
```

- 7** To get a sense of what the data is, view the data for January, which is in column 2. Type

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

MATLAB returns

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

- 8** Get the size of the matrix containing the fetched data set, assigning the dimensions to m and n. In a later step, you use these values to compute the monthly totals. Type

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

MATLAB returns

```
m =  
    10  
n =  
    13
```

9 Compute the monthly totals by typing

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

where `tmp` is the sales volume for all products in a given month `i`, and `monthly` is the total sales volume of all products for the month `i`.

For example, when `i` 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.

To see the result, type

```
monthly
```

MATLAB returns

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

- 10** Create a string array containing the column names into which you are inserting the data. In a later step, we insert the data into the `salesTotal` column of the `yearlySales` table. The `yearlySales` table contains no data. Here we assign the array to the variable `colnames`. Type

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

- 11** Insert the data into the `yearlySales` table by typing
- ```
insert(conn, 'yearlySales', colnames, monthly)
```

- 12** View the `yearlySales` table in the tutorial database to be sure the data was imported correctly.

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

Record: 1 of 12

- 13** Close the cursor and database connection. Type

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

## Accessing Metadata

In this example, you access information about the database, which is called the *metadata*. You use these Database Toolbox functions:

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

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

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

MATLAB returns information about the database object.

```
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'
```

**2** To view additional information about the database, you first construct a database metadata object using the `dmd` function. Type

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

MATLAB returns the handle (identifier) for the metadata object.

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

- 3** To view a list of properties associated with the database, use the `get` function for the metadata object you just created, `dbmeta`.

```
v = get(dbmeta)
```

MATLAB returns a long list of properties associated with the database.

```
v =  
    AllProceduresAreCallable: [1x1 logical]  
    AllTablesAreSelectable: [1x1 logical]  
DataDefinitionCausesTransactionCommit: [1x1 logical]  
DataDefinitionIgnoredInTransactions: [1x1 logical]  
    DoesMaxRowSizeIncludeBlobs: [1x1 logical]  
        Catalogs: {1x1 cell}  
    CatalogSeparator: [1x1 char]  
    CatalogTerm: [1x8 char]  
    DatabaseProductName: [1x6 char]  
    DatabaseProductVersion: [1x10 char]  
DefaultTransactionIsolation: [1x1 double]  
    DriverMajorVersion: [1x1 double]  
    DriverMinorVersion: [1x1 double]  
    DriverName: [1x31 char]  
    DriverVersion: [1x19 char]  
    ExtraNameCharacters: [1x29 char]  
IdentifierQuoteString: [1x1 char]  
    IsCatalogAtStart: [1x1 logical]  
MaxBinaryLiteralLength: [1x1 double]  
    MaxCatalogNameLength: [1x1 double]  
    MaxCharLiteralLength: [1x1 double]  
    MaxColumnNameLength: [1x1 double]  
    MaxColumnsInGroupBy: [1x1 double]  
    MaxColumnsInIndex: [1x1 double]  
    MaxColumnsInOrderBy: [1x1 double]  
    MaxColumnsInSelect: [1x1 double]  
    MaxColumnsInTable: [1x1 double]  
    MaxConnections: [1x1 double]  
    MaxCursorNameLength: [1x1 double]  
    MaxIndexLength: [1x1 double]  
MaxProcedureNameLength: [1x1 double]  
    MaxRowSize: [1x1 double]  
    MaxSchemaNameLength: [1x1 double]
```



```

MaxStatementLength: [1x1 double]
  MaxStatements: [1x1 double]
MaxTableNameLength: [1x1 double]
  MaxTablesInSelect: [1x1 double]
  MaxUserNameLength: [1x1 double]
  NumericFunctions: [1x73 char]
  ProcedureTerm: [1x5 char]
    Schemas: {}
    SchemaTerm: ''
  SearchStringEscape: [1x1 char]
  SQLKeywords: [1x461 char]
  StringFunctions: [1x91 char]
  StoresLowerCaseIdentifiers: [1x1 logical]
  StoresLowerCaseQuotedIdentifiers: [1x1 logical]
  StoresMixedCaseIdentifiers: [1x1 logical]
  StoresMixedCaseQuotedIdentifiers: [1x1 logical]
  StoresUpperCaseIdentifiers: [1x1 logical]
  StoresUpperCaseQuotedIdentifiers: [1x1 logical]
  SystemFunctions: ''
  TableTypes: {4x1 cell}
  TimeDateFunctions: [1x111 char]
  TypeInfo: {16x1 cell}
  URL: [1x23 char]
  UserName: [1x5 char]
NullPlusNonNullIsNull: [1x1 logical]
  NullsAreSortedAtEnd: [1x1 logical]
  NullsAreSortedAtStart: [1x1 logical]
  NullsAreSortedHigh: [1x1 logical]
  NullsAreSortedLow: [1x1 logical]
  UsesLocalFilePerTable: [1x1 logical]
  UsesLocalFiles: [1x1 logical]

```

You can see much of the information in the list directly, for example, the `UserName`, which is `'admin'`.

- 4** Some information is too long to fit in the field's display area and instead the size of the information in the field is reported. For example, the Catalogs element is shown as a {1x1 cell}. To view the actual Catalog information, type

```
v.Catalogs
```

MATLAB returns

```
ans =  
      'D:\matlab\toolbox\database\dbdemos\tutorial'
```

For more information about the database metadata properties returned by `get`, see the methods of the `DatabaseMetaData` object at <http://java.sun.com/products/jdk/1.2/docs/api/java/sql/package-summary.html>.

- 5** To see the properties that this database supports, use the `supports` function. Type

```
a = supports(dbmeta)
```

## MATLAB returns

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 means the database supports that property, while a 0 means the database does not support that property. For the above example, the `GroupBy` property has a value of 1, meaning the database supports the SQL group by feature.

For more information about the properties supported by the database, see the methods of the `DatabaseMetaData` object at <http://java.sun.com/products/jdk/1.2/docs/api/java/sql/package-summary.html>.

- 6 There are other Database Toolbox functions you can use to access additional database metadata. For example, to retrieve the names of the tables in a catalog in the database, use the `tables` function. Type

```
t = tables(dbmeta, 'tutorial')
```

where `dbmeta` is the name of the database metadata object you created for the database using `dmd` in step 2, and `tutorial` is the name of the catalog for which you want to retrieve table names. (You retrieved catalog names in step 4.)

MATLAB returns the names and types for each table.

```
t =
  'MSysACEs'           'SYSTEM TABLE'
  'MSysIMEXColumns'   'SYSTEM TABLE'
  'MSysIMEXSpecs'     'SYSTEM TABLE'
  'MSysModules'       'SYSTEM TABLE'
  'MSysModules2'      'SYSTEM TABLE'
  'MSysObjects'       'SYSTEM TABLE'
  'MSysQueries'       'SYSTEM TABLE'
  'MSysRelationships' 'SYSTEM TABLE'
  'inventoryTable'    'TABLE'
  'productTable'      'TABLE'
  'salesVolume'       'TABLE'
  'suppliers'         'TABLE'
  'yearlySales'       'TABLE'
  'display'           'VIEW'
```

Two of these tables were used in the previous example: `salesVolume` and `yearlySales`.

For a list of all of the database metadata functions, see “Database Metadata Object” on page 4-5. Some databases do not support all of these functions.

**7** Close the database connection. Type

```
close(conn)
```

## Resultset Metadata Object

Similar to the `dmd` function are the `resultset` and `rsmd` functions. Use `resultset` to create a resultset object for a cursor object that you created using `exec` or `fetch`. You can then get properties of the resultset object, create a resultset metadata object using `rsmd` and get its properties, or make calls to the resultset object using your own Java-based applications. For more information, see the reference pages for `resultset` and `rsmd`, or see the lists of related functions, “Resultset Object” and “Resultset Metadata Object” on page 4-6.

## Performing Driver Functions

This example demonstrates how to create database driver and drivermanager objects so that you can get and set the object properties. You use these Database Toolbox functions:

- drivermanager
- driver
- get
- isdriver
- set

---

**Note** There is no equivalent M-file demo to run because the example relies on a specific PC to JDBC connection and database. Your configuration will be different than the one in this example so you cannot run these examples exactly as written. Instead, use values for your own system. See your database administrator for address 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 for a specified database URL string of the form `jdbc:<subprotocol>:<subname>`. For example, type

```
d = driver('jdbc:oracle:thin:@144.212.123.24:1822:')
```

MATLAB returns the handle (identifier) for the driver object.

```
d =  
    DriverHandle: [1x1 oracle.jdbc.driver.OracleDriver]
```

- 3** To get properties of the driver object, type

```
v = get(d)
```

MATLAB returns information about the driver's versions.

```
v =  
    MajorVersion: 1  
    MinorVersion: 0
```

- 4** To determine if `d` is a valid JDBC driver object, type

```
isdriver(d)
```

MATLAB returns

```
ans =  
     1
```

which means `d` is a valid JDBC driver object. Otherwise, MATLAB would have returned a 0.

- 5** To set and get properties for all drivers, first create a `drivermanager` object using the `drivermanager` function. Type

```
dm = drivermanager
```

`dm` is the `drivermanager` object.

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

```
v = get(dm)
```

MATLAB returns

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

- 7** To set the `LoginTimeout` value to 10 for all drivers loaded during this session, type

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

Verify the value by typing

```
v = get(dm)
```

MATLAB returns

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

The next time you connect to a database, the `LoginTimeout` value will be 10.

For example, type

```
conn = database('SampleDB','','');  
logintimeout
```

MATLAB returns

```
ans =  
    10
```

For a list of all the driver object functions, see “Driver Object” and “Drivermanager Object” on page 4-6.



## About Objects and Methods for the Database Toolbox

The Database Toolbox is an object-oriented application. The toolbox has the following objects:

- Cursor
- Database
- Database metadata
- Driver
- Drivermanager
- Resultset
- Resultset metadata

Each object has its own method directory, which begins with an @ sign, in the `$matlabroot\toolbox\database\database` directory. The methods for operating on a given object are the M-file functions in the object's directory.

You can use the Database Toolbox with no knowledge of or interest in its object-oriented implementation. But for those that are interested, some of its useful aspects follow:

- You use constructor functions to create objects, such as running the `fetch` function to create a cursor object containing query results. MATLAB returns not only the object but stored information about the object. Since objects are structures in MATLAB, you can easily view the elements of the returned object.

As an example, if you create a cursor object `curs` using the `fetch` function, MATLAB returns

```
curs =  
    Attributes: []  
           Data: {10x1 cell}  
 DatabaseObject: [1x1 database]  
           RowLimit: 0  
           SQLQuery: 'select country from customers'  
           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: com.mathworks.toolbox.database.fetchTheData]
```

You can easily access information about the cursor object, including the results, which are in the `Data` element of the cursor object. To view the contents of the element, which is a 10-by-1 cell array in this example, you type

```
curs.Data
```

MATLAB returns

```
ans =  
    'Germany'  
    'Mexico'  
    'Mexico'  
    'UK'  
    'Sweden'  
    'Germany'  
    'France'  
    'Spain'  
    'France'
```

- Objects allow the use of overloaded functions. For example, to view properties of objects in the Database Toolbox, you use the `get` function, regardless of the object. This means you only have to remember one function, `get`, rather than having to remember specific functions for each object. The properties you retrieve with `get` differ, depending on the object, but the function itself always has the same name and argument syntax.

- You can write your own methods, as M-files, to operate on the objects in the Database Toolbox. For more information, see “MATLAB Classes and Objects” in the MATLAB documentation.

## Working with Cell Arrays in MATLAB

When you import data from a database into MATLAB, the data is stored as a numeric matrix, a structure, or a MATLAB cell array, depending on the data return format preference you specified using `setdbprefs` or the **Database Toolbox Preferences** dialog box.

Once the data is in MATLAB, you can use MATLAB functions to work with it. Because some users are unfamiliar with cell arrays, this section provides a few simple examples of how to work with the cell array data type in MATLAB:

- “Viewing Cell Array Data Returned from a Query” on page 3-37
- “Viewing Elements of Cell Array Data” on page 3-39
- “Performing Functions on Cell Array Data” on page 3-41
- “Creating Cell Arrays for Exporting Data from MATLAB” on page 3-41

For more information on using cell arrays, see “Structures and Cell Arrays” in the MATLAB documentation.

## Viewing Cell Array Data Returned from a Query

### Viewing Query Results

- 1 How you view query results depends on if you imported the data using the `fetch` function or if you used the Visual Query Builder.
  - If you imported data using the `fetch` function, MATLAB returns, for example (see “Exporting Data from MATLAB to a New Record in a Database” on page 3-11)

```
curs =
  Attributes: []
           Data: [3x1 double]
 DatabaseObject: [1x1 database]
      RowLimit: 0
   SQLQuery: 'select freight from orders'
      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]
```

The retrieved data is in the field **Data**. To view it, type

```
curs.Data
```

Alternatively, you can assign the data to a variable, for example, `A`, by typing

```
A = curs.Data
```

and then view it by typing `A`.

- If you imported data using the Visual Query Builder, you assign the results to the workspace variable, which is `A` in this example, using the Visual Query Builder. To see the data, type the workspace variable name at the MATLAB prompt in the **Command Window**, for example, type `A`.

**2** MATLAB displays the data in the **Command Window**, for example

```
A =  
    32.3800  
    11.6100  
    65.8300
```

### Viewing Results with Multiple Columns

If the query results consist of multiple columns, you can view all the results for a single column using a colon (:). See the example in “Exporting Multiple New Records from MATLAB” on page 3-19. For example, you view the results of column 2 by typing

```
A(:,2)
```

or if you used `fetch`, you can also view it by typing

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

MATLAB returns the data in column 2, for example

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

### Expanding Results

If the results do not fit in the display space available, MATLAB expresses them as an array. If for example, MATLAB returns these query results.

```
B =  
 [122]    'Virgina Power'  
 [123]    'North Land Trading'  
 [124]                [1x20 char]  
 [125]    'Bush Pro Shop'
```

You can see the data in rows 1, 2, and 4, but the second column in row 3 is expressed as an array because the results are too long to display.

To view the contents of the second column in the third row, type

```
B(3,2)
```

or if you used `fetch`, you can also view it by typing

```
curs.Data(3,2)
```

MATLAB returns

```
ans =  
    'The Ristuccia Center'
```

## Viewing Elements of Cell Array Data

In these examples, the `curs.Data` notation is not used and instead the examples assume you assigned `curs.Data` to a variable. If you do not assign `curs.Data` to a variable, then just substitute `curs.Data` for the variable name in the examples.

This example is the same as that in “Exporting Data from MATLAB to a New Record in a Database” on page 3-11, but the `DataReturnFormat` is set to `cellarray`.

```
A =  
    [32.3800]  
    [11.6100]  
    [65.8300]
```

## Viewing a Single Element as a Numeric Value

To view the first element of `A`, type

```
A(1,1)
```

MATLAB returns

```
ans =  
    [32.3800]
```

The result is not numeric but instead is an element in a cell array. You cannot perform numeric operations on cell array data.

To retrieve the first element as a numeric value, enclose it in curly braces. For example, type

```
A{1,1}
```

MATLAB returns

```
ans =  
32.3800
```

This result is numeric and therefore you can perform numeric operations on it.

#### **Viewing an Entire Column or Row as a Numeric Vector**

To retrieve the data in an entire column or row of a cell array as a numeric vector, use colons within the curly braces. You then assign the results to a matrix by enclosing them in square brackets. For example, to retrieve all the data in column 1, type

```
AA=[A{: ,1}]'
```

MATLAB returns

```
AA =  
32.3800  
11.6100  
65.8300
```

You can also retrieve the contents using the `celldisp` function. For example, type

```
celldisp(A)
```

MATLAB returns

```
A{1} =  
32.3800  
  
A{2} =  
11.6100  
  
A{3} =  
65.8300
```



## Performing Functions on Cell Array Data

To perform certain MATLAB functions directly on cell arrays, you need to extract the contents of the cell array as numeric data.

For example, to compute the sum of the elements in the cell array `A`, type

```
sum([A{:}])
```

MATLAB returns

```
ans =  
    109.8200
```

## Creating Cell Arrays for Exporting Data from MATLAB

If you use the `insert` or `update` functions to export data from MATLAB to a database and need to include data in a cell array, such as column names, use the following techniques.

### Enclosing Data in Curly Braces

One way to put data in a cell array is by enclosing the data in curly braces, with rows separated by semicolons, and elements within a row separated by commas.

For example, to identify the column names in an `insert` function, use curly braces as follows.

```
insert(conn, 'Growth', {'Date','Average'}, insertdata)
```

You can also insert the data itself using curly braces. For example, to insert `A` and `avgA`, and `B` and `avgB`, into the `Date` and `Average` columns of the `Growth` table, use the `insert` function as follows.

```
insert(conn, 'Growth', {'Date','Average'}, {A, avgA; B, avgB})
```

### Assigning Cell Array Elements

To put data into a cell array element, enclose it in curly braces. For example, if you have one row containing two values you want to export, `A` and `meanA`, put them in cell array `exdata`, which you will export. Type

```
exdata(1,1) = {A};  
exdata(1,2) = {meanA};
```

Alternatively, you can assign values to `exdata` in one step by typing

```
exdata = {A,meanA}
```

To export the data `exdata`, use the `insert` function as follows.

```
insert(conn, 'Growth', colnames, exdata)
```

#### **Converting a Numeric Matrix to a Cell Array**

If you want to export data containing numeric and string values, you need to export it as a cell array. As an example, you will export a cell array, `exdata`, whose first column already contains the names of the twelve months. You have calculated the total sales figures for each month and the results are in the numeric matrix `monthly`. To assign the values in `monthly` to the second column of the cell array `exdata`, convert the numeric matrix `monthly` to a cell array `exdata` using the `num2cell`. Type

```
exdata(:,2) = num2cell(monthly);
```

`num2cell` takes the data in `monthly` and assigns each row to the second column in the cell array, `exdata`.

# Function Reference

---

### Functions—By Category

The following tables group Database Toolbox functions by category:

- “General” on page 4-3
- “Database Connection” on page 4-3
- “SQL Cursor” on page 4-3
- “Importing Data into MATLAB from a Database” on page 4-4
- “Exporting Data from MATLAB to a Database” on page 4-4
- “Database Metadata Object” on page 4-5
- “Driver Object” on page 4-6
- “Drivermanager Object” on page 4-6
- “Resultset Object” on page 4-6
- “Resultset Metadata Object” on page 4-6
- “Visual Query Builder” on page 4-7

## General

`logintimeout` Set or get time allowed to establish database connection.  
`setdbprefs` Set or get time allowed to establish database connection.

## Database Connection

`close` Close database connection.  
`database` Connect to database.  
`get` Get property of database connection.  
`isconnection` Detect if database connection is valid.  
`isreadonly` Detect if database connection is read-only.  
`ping` Get status information about database connection.  
`set` Set properties for database connection.  
`setdbprefs` Set or get time allowed to establish database connection.  
`sql2native` Convert JDBC SQL grammar to system's native SQL grammar.

## SQL Cursor

`close` Close cursor.  
`exec` Execute SQL statement and open cursor.  
`get` Get property of cursor object.  
`querytimeout` Get time allowed for a database SQL query to succeed.  
`set` Set `RowLimit` for cursor fetch.

### **Importing Data into MATLAB from a Database**

<code>attr</code>	Get attributes of columns in fetched data set.
<code>cols</code>	Get number of columns in fetched data set.
<code>columnnames</code>	Get names of columns in fetched data set.
<code>fetch</code>	Import data into MATLAB.
<code>rows</code>	Get number of rows in fetched data set.
<code>width</code>	Get field size of column in fetched data set.

### **Exporting Data from MATLAB to a Database**

<code>commit</code>	Make database changes permanent.
<code>insert</code>	Export MATLAB data into database table.
<code>rollback</code>	Undo database changes.
<code>update</code>	Replace data in database table with data from MATLAB.

## Database Metadata Object

<code>bestrowid</code>	Get database table unique row identifier.
<code>columnprivileges</code>	Get database column privileges.
<code>columns</code>	Get database table column names.
<code>crossreference</code>	Get information about primary and foreign keys.
<code>dmd</code>	Construct database metadata object.
<code>exportedkeys</code>	Get information about exported foreign keys.
<code>get</code>	Get database metadata properties.
<code>importedkeys</code>	Get information about imported foreign keys.
<code>indexinfo</code>	Get indices and statistics for database table.
<code>primarykeys</code>	Get primary key information for database table or schema.
<code>procedurecolumns</code>	Get catalog's stored procedure parameters and result columns.
<code>procedures</code>	Get catalog's stored procedures.
<code>supports</code>	Detect if property is supported by database metadata object.
<code>tableprivileges</code>	Get database table privileges.
<code>tables</code>	Get database table names.
<code>versioncolumns</code>	Get automatically updated table columns.

### Driver Object

<code>driver</code>	Construct database driver object.
<code>get</code>	Get database driver properties.
<code>isdriver</code>	Detect if driver is a valid JDBC driver object.
<code>isjdbc</code>	Detect if driver is JDBC-compliant.
<code>isurl</code>	Detect if the database URL is valid.
<code>register</code>	Load database driver.
<code>unregister</code>	Unload database driver.

### Drivermanager Object

<code>drivermanager</code>	Construct database drivermanager object.
<code>get</code>	Get database drivermanager properties.
<code>set</code>	Set database drivermanager properties.

### ResultSet Object

<code>clearwarnings</code>	Clear the warnings for the resultset.
<code>close</code>	Close resultset object.
<code>get</code>	Get resultset properties.
<code>isnullcolumn</code>	Detect if last record read in resultset was NULL.
<code>namecolumn</code>	Map resultset column name to resultset column index.
<code>resultset</code>	Construct resultset object.

### ResultSet Metadata Object

<code>get</code>	Get resultset metadata properties.
<code>rsmd</code>	Construct resultset metadata object.



## Visual Query Builder

<code>confds</code>	Configure data source for use with Visual Query Builder (JDBC only).
<code>querybuilder</code>	Start visual SQL query builder.

## Functions—Alphabetical List

attr	4-10
bestrowid	4-12
clearwarnings	4-13
close	4-14
cols	4-16
columnnames	4-17
columnprivileges	4-18
columns	4-20
commit	4-22
confds	4-23
crossreference	4-24
database	4-27
dmd	4-30
driver	4-31
drivermanager	4-32
exec	4-33
exportedkeys	4-36
fetch	4-39
get	4-43
importedkeys	4-51
indexinfo	4-54
insert	4-56
isconnection	4-60
isdriver	4-61
isjdbc	4-62
isnullcolumn	4-63
isreadonly	4-65
isurl	4-66
logintimeout	4-67
namecolumn	4-70
ping	4-71
primarykeys	4-73
procedurecolumns	4-75
procedures	4-77
querybuilder	4-79

querytimeout .....	4-80
register .....	4-81
resultset .....	4-82
rollback .....	4-83
rows .....	4-84
rsmd .....	4-85
set .....	4-86
setdbprefs .....	4-92
sql2native .....	4-101
supports .....	4-102
tableprivileges .....	4-104
tables .....	4-105
unregister .....	4-106
update .....	4-107
versioncolumns .....	4-109
width .....	4-111

# attr

---

**Purpose** Get 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 specified 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 it in a cell array. Use `attributes(colnum)` to display the attributes for column `colnum`.

The returned attributes are listed 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> , the data cannot be overwritten
<code>nullable</code>	If <code>true</code> , the data can be NULL
<code>Message</code>	Error message returned by fetch

## Examples

### Example 1—Get Attributes for One Column

Get the 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

Get the column attributes for curs, and assign them to attributes.

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

View the attributes of column 4.

```
attributes(4)
```

MATLAB returns the attributes of column 4.

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

## See Also

cols, columnnames, columns, dmd, fetch, get, tables, width

# bestrowid

---

**Purpose** Get database table unique row identifier

**Syntax**  
`b = bestrowid(dbmeta, 'cata', 'sch')`  
`b = bestrowid(dbmeta, 'cata', 'sch', 'tab')`

**Description** `b = bestrowid(dbmeta, 'cata', 'sch')` determines and returns the optimal set of columns in a table that uniquely identifies a row, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`b = bestrowid(dbmeta, 'cata', 'sch', 'tab')` determines and returns the optimal set of columns that uniquely identifies a row in table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

**Examples** Type  
`b = bestrowid(dbmeta, 'msdb', 'geck', 'builds')`

MATLAB returns

```
b =  
    'build_id'
```

In this example:

- `dbmeta` is the database metadata object.
- `msdb` is the catalog `cata`.
- `geck` is the schema `sch`.
- `builds` is the table `tab`.

The results is `build_id`, which means that every entry in the `build_id` column is unique and can be used to identify the row.

**See Also** `columns`, `dmd`, `get`, `tables`

<b>Purpose</b>	Clear warnings for database connection or resultset
<b>Syntax</b>	<code>clearwarnings(conn)</code> <code>clearwarnings(rset)</code>
<b>Description</b>	<p><code>clearwarnings(conn)</code> clears the warnings reported for the database connection object <code>conn</code>, which was created using <code>database</code>.</p> <p><code>clearwarnings(rset)</code> clears the warnings reported for the resultset object <code>rset</code>, which was created using <code>resultset</code>.</p> <p>For command line help on <code>clearwarnings</code>, use the overloaded methods:</p> <pre>help database/clearwarnings help resultset/clearwarnings</pre>
<b>Examples</b>	<code>clearwarnings(conn)</code> NULLS reported warnings for the database connection object <code>conn</code> , which was created using <code>conn = database(...)</code> .
<b>See Also</b>	<code>database</code> , <code>get</code> , <code>resultset</code>

# close

---

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

**Syntax** `close(object)`

**Description** `close(object)` closes `object`, freeing up associated resources. Following are the allowable objects for `close`.

<b>Object</b>	<b>Description</b>	<b>Action Performed by <code>close(object)</code></b>
<code>conn</code>	Database connection object created using <code>database</code>	closes <code>conn</code>
<code>curs</code>	Cursor object created using <code>exec</code> or <code>fetch</code>	closes <code>curs</code>
<code>rset</code>	Resultset object defined using <code>resultset</code>	closes <code>rset</code>

Database connections, cursors, and resultsets remain open until you close them using the `close` function. Always close a cursor, connection, or resultset when you finish using it so that MATLAB stops reserving memory for it. Also, most databases limit the number of cursors and connections that can be open at one time.

If you terminate a MATLAB session while cursors and connections are open, MATLAB closes them, but your database might not free up the connection or cursor. Therefore, always close connections and cursors when you finish using them.

Close a cursor before closing the connection used for that cursor.

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

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



**Examples**

To close the cursor `curs` and the connection `conn`, type

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

**See Also**

database, exec, fetch, resultset

# cols

---

**Purpose** Get 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** This example shows that there are three columns in the fetched data set, `curs`.

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

```
numcols =  
3
```

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

**Purpose** Get names of columns in fetched data set

**Syntax** `colnames = columnnames(curs)`

**Description** `colnames = columnnames(curs)` returns the column names in the fetched data set `curs`. The column names are returned as a single string vector.

**Examples** The fetched data set `curs`, contains three columns having the names shown.

```
colnames = columnnames(curs)
```

```
colnames =  
  'Address', 'City', 'Country'
```

**See Also** `attr`, `cols`, `columnprivileges`, `columns`, `fetch`, `get`, `width`

# columnprivileges

---

**Purpose** Get 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 the list of privileges for all columns in table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`lp = columnprivileges(dbmeta, 'cata', 'sch', 'tab', 'l')` returns the list of privileges for column `l`, in the table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

## Examples

Type

```
lp = columnprivileges(dbmeta,'msdb','geck','builds','build_id')
```

MATLAB returns

```
lp =  
    'builds'    'build_id'    {1x4 cell}
```

In this example:

- `dbmeta` is the database metadata object.
- `msdb` is the catalog `cata`.
- `geck` is the schema `sch`.
- `builds` is the table `tab`.
- `build_id` is the column name.

The results show:

- The table name, `builds`, in column 1.
- The column name, `build_id`, in column 2.
- The column privileges, `lp`, in column 3.

To view the contents of the third column in `lp`, type

```
lp{1,3}
```

MATLAB returns the column privileges for the `build_id` column.

```
ans =  
    'INSERT'    'REFERENCES'    'SELECT'    'UPDATE'
```

### See Also

`cols`, `columns`, `columnnames`, `dmd`, `get`

# columns

---

**Purpose** Get 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 the list of all column names in the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`l = columns(dbmeta, 'cata', 'sch')` returns the list of all column names in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`l = columns(dbmeta, 'cata', 'sch', 'tab')` returns the list of columns for the table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

## Examples

Type

```
l = columns(dbmeta, 'orcl', 'SCOTT')
```

MATLAB returns

```
l =
    'BONUS'      {1x4 cell}
    'DEPT'       {1x3 cell}
    'EMP'        {1x8 cell}
    'SALGRADE'   {1x3 cell}
    'TRIAL'      {1x3 cell}
```

In this example:

- `dbmeta` is the database metadata object.
- `orcl` is the catalog `cata`.
- `SCOTT` is the schema `sch`.

The results show the names of the five tables and a cell array containing the column names in the tables.

To see the column names for the BONUS table, type

```
l{1,2}
```

MATLAB returns

```
ans =  
    'ENAME'    'JOB'    'SAL'    'COMM'
```

which are the column names in the BONUS table.

## See Also

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

# commit

---

**Purpose** Make database changes permanent

**Syntax** `commit(conn)`

**Description** `commit(conn)` makes permanent the changes made via `insert` or `update` to the database connection `conn`. The `commit` function commits all changes made since the last `commit` or `rollback` function was run, or the last `exec` function that performed a `commit` or `rollback`. The `AutoCommit` flag for `conn` must be `off` to use `commit`.

**Examples** Ensure the `AutoCommit` flag for connection `conn` is `off` by typing

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

MATLAB returns

```
ans =  
off
```

Insert the data contained in `exdata` into the columns `DEPTNO`, `DNAME`, and `LOC`, in the table `DEPT` for the data source `conn`. Type

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

Commit the data inserted in the database by typing

```
commit(conn)
```

The data is added to the database.

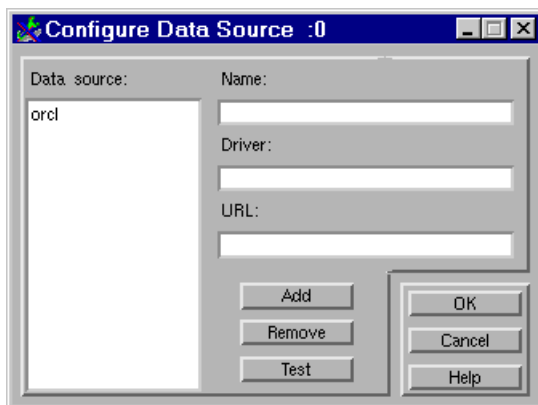
**See Also** `database`, `exec`, `get`, `insert`, `rollback`, `update`



**Purpose** Configure data source for Visual Query Builder (JDBC on UNIX only)

**Syntax** confds

**Description** confds displays the **Configure Data Source** dialog box, from which you add and remove data sources. Use confds on a UNIX platform if you connect to databases via JDBC drivers and want to use the Visual Query Builder. (You cannot use the Visual Query Builder on a Windows platform with a JDBC driver.) To add and remove data sources for connections that use ODBC drivers, see “Setting Up a Data Source” on page 1-7.



- 1 Complete the **Name**, **Driver**, and **URL** fields. For example:  
**Name:** orcl  
**Driver:** oracle.jdbc.driver.OracleDriver  
**URL:** jdbc:oracle:thin:@144.212.123.24:1822:
- 2 Click **Add** to add the data source.
- 3 Click **Test** to establish a test connection to the data source. You are prompted to supply a username and password if the database requires it.
- 4 Click **OK** to save the changes and close the **Configure Data Source** dialog box.

To remove a data source, select it, click **Remove**, and click **OK**.

# crossreference

---

**Purpose** Get 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. Specifically, the information is for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`. The primary key information is for the table `ptab`, in the primary schema `psch`, of the primary catalog `pcata`. The foreign key information is for the foreign table `ftab`, in the foreign schema `fsch`, of the foreign catalog `fcata`.

**Examples** Type

```
f = crossreference(dbmeta, 'orcl', 'SCOTT', 'DEPT', ...
    'orcl', 'SCOTT', 'EMP')
```

MATLAB returns

```
f =
Columns 1 through 7
    'orcl'    'SCOTT'    'DEPT'    'DEPTNO'    'orcl'    'SCOTT'    'EMP'
Columns 8 through 13
    'DEPTNO'    '1'    'null'    '1'    'FK_DEPTNO'    'PK_DEPT'
```

In this example:

- `dbmeta` is the database metadata object.
- `orcl` is the catalog `pcata` and the catalog `fcata`.
- `SCOTT` is the schema `psch` and the schema `fsch`.
- `DEPT` is the table `ptab` that contains the referenced primary key.
- `EMP` is the table `ftab` that contains the foreign key.

The results show the primary and foreign key information.

Column	Description	Value
1	Catalog containing primary key, referenced by foreign imported key	orcl
2	Schema containing primary key, referenced by foreign imported key	SCOTT
3	Table containing primary key, referenced by foreign imported key	DEPT
4	Column name of primary key, referenced by foreign imported key	DEPTNO
5	Catalog that has foreign key	orcl
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 foreign key	1
10	Update rule, that is, what happens to the foreign key when the primary key is updated.	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 key. The table DEPT contains a primary key DEPTNO that is referenced by the field DEPTNO in the table EMP. DEPTNO in the EMP table is a foreign key.

## crossreference

---

For a description of the codes for update and delete rules, see <http://java.sun.com/products/jdk/1.2/docs/api/java/sql/package-summary.html> for the DatabaseMetaData object property `getCrossReference`.

### See Also

`dmd`, `exportedkeys`, `get`, `importedkeys`, `primarykeys`

**Purpose** Connect to database

**Syntax**

```
conn = database('datasourcename', 'username', 'password')
conn = database('databasename', 'username', 'password',
    'driver', 'databaseurl')
```

**Description**

`conn = database('datasourcename', 'username', 'password')` connects a MATLAB session to a database via an ODBC driver, returning the connection object to `conn`. The data source to which you are connecting is `datasourcename`. You must have previously set up the data source—for instructions, see “Setting Up a Data Source”. `username` and `password` are the username and/or password required to connect to the database. If you do not need a username or a password to connect to the database, use empty strings as the arguments.

`conn = database('databasename', 'username', 'password', 'driver', 'databaseurl')` connects a MATLAB session to a database, `databasename`, via the specified JDBC driver, returning the connection object to `conn`. The username and/or password required to connect to the database are `username` and `password`. If you do not need a username or a password to connect to the database, use empty strings as the arguments. `databaseurl` is the JDBC URL object, `jdbc:subprotocol:subname`. The subprotocol is a database type, such as `oracle`. The subname may contain other information used by driver, such as the location of the database and/or a port number. The subname may take the form `//hostname:port/databasename`. Find the correct driver name and `databaseurl` format in the driver manufacturer’s documentation.

If `database` establishes a connection, MATLAB returns information about the connection object.

```
Instance: 'SampleDB'
UserName: ''
Driver: []
URL: []
Constructor: [1x1
    com.mathworks.toolbox.database.databaseConnect]
Message: []
Handle: [1x1 sun.jdbc.odbc.JdbcOdbcConnection]
TimeOut: 0
AutoCommit: 'off'
Type: 'Database Object'
```

Use `logintimeout` before you use `database` to specify the maximum amount of time for which `database` tries to establish a connection.

You can have multiple `database` connections open at one time.

After connecting to a database, use the `ping` function to view status information about the connection, and use `dmd`, `get`, and `supports` to view properties of `conn`.

The `database` connection stays open until you close it using the `close` function. Always close a connection after you finish using it.

## Examples

### Example 1—Establish ODBC Connection

To connect to an ODBC data source called `Pricing`, where the database has a user `mike` and a password `bravo`, type

```
conn = database('Pricing', 'mike', 'bravo');
```

### Example 2—Establish ODBC Connection Without Username and Password

To connect to an ODBC data source `SampleDB`, where a username and password are not needed, use empty strings in place of those arguments. Type

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

### Example 3—Establish JDBC Connection

In this JDBC connection example, the database is `oracle`, the username is `scott`, and the password is `tiger`. The `oci7` JDBC driver name is `oracle.jdbc.driver.OracleDriver` and the URL that specifies the location of the database server is `jdbc:oracle:oci7`.

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

The JDBC name and URL take different forms for different databases, as shown in the examples in the following table.

Database	JDBC Driver and Database URL
Oracle oci7 drivers	JDBC driver: <code>oracle.jdbc.driver.OracleDriver</code> Database URL: <code>jdbc:oracle:oci7:</code>
Oracle oci8 drivers	JDBC driver: <code>oracle.jdbc.driver.OracleDriver</code> Database URL: <code>jdbc:oracle:oci8:@111.222.333.44:1521:</code>
Oracle thin drivers	JDBC driver: <code>oracle.jdbc.driver.OracleDriver</code> Database URL: <code>jdbc:oracle:thin:@144.212.123.24:1822:</code>
MySQL	JDBC driver: <code>twz1.jdbc.mysql.jdbcMySQLDriver</code> Database URL: <code>jdbc:z1MySQL://natasha:3306/metrics</code>
Sybase jConnect	JDBC driver: <code>com.sybase.jdbc.SybDriver</code> Database URL: <code>jdbc:sybase:Tds:yourmachinename:portnumber/</code>

For the Oracle thin drivers example, in the database URL `jdbc:oracle:thin:@144.212.123.24:1822`, the target machine that the database server resides on is `144.212.123.24`, and the port number is `1822`.

## See Also

`close`, `dmd`, `get`, `isconnection`, `isreadonly`, `logintimeout`, `ping`, `supports`

# dmd

---

**Purpose** Construct database metadata object

**Syntax** `dbmeta = dmd(conn)`

**Description** `dbmeta = dmd(conn)` constructs a database metadata object for the database connection `conn`, which was created using `database`. Use `get` and `supports` to obtain properties of `dbmeta`. Use `dmd` and `get(dbmeta)` to obtain information you need about a database, such as the database table names to retrieve data using `exec`.

For a list of other functions you can perform on `dbmeta`, type

```
help dmd/Contents
```

**Examples** `dbmeta = dmd(conn)` creates the database metadata object `dbmeta` for the database connection `conn`.

`v = get(dbmeta)` lists the properties of the database metadata object.

**See Also** `columns`, `database`, `get`, `supports`, `tables`



**Purpose** Construct database driver object

**Syntax** `d = driver('s')`

**Description** `d = driver('s')` constructs a database driver object `d`, from `s`, where `s` is a database URL string of the form `jdbc:odbc:<name>` or `<name>`. The driver object `d` is the first driver that recognizes `s`.

**Examples** `d = driver('jdbc:odbc:thin:@144.212.123.24:1822:')` creates driver object `d`.

**See Also** `get`, `isdriver`, `isjdbc`, `isurl`, `register`

# drivermanager

---

**Purpose** Construct database drivermanager object

**Syntax** `dm = drivermanager`

**Description** `dm = drivermanager` constructs a database drivermanager object. You can then use `get` and `set` to obtain and change the properties of `dm`, which are the properties for all loaded database drivers as a whole.

**Examples** `dm = drivermanager` creates the database drivermanager object `dm`.  
`get(dm)` returns the properties of the drivermanager object `dm`.

**See Also** `get`, `register`, `set`

**Purpose** Execute SQL statement and open cursor

**Syntax** `curs = exec(conn, 'sqlquery')`

**Description** `curs = exec(conn, 'sqlquery')` executes the valid SQL statement `sqlquery`, against the database connection `conn`, and opens a cursor. Running `exec` returns the cursor object to the variable `curs`, and returns information about the cursor object. The `sqlquery` argument can also be a stored procedure for that database connection.

### Notes

- After opening a cursor, use `fetch` to import data from the cursor. Use `resultset`, `rsmd`, and `statement` to get properties of the cursor.
- Use `querytimeout` to determine the maximum amount of time for which `exec` will try to complete the SQL statement.
- You can have multiple cursors open at one time.
- A cursor stays open until you close it using the `close` function. Always close a cursor after you finish using it.

### Examples

#### Example 1 – Select All Data from Database Table

Select all data from the `customers` table accessed via the database connection, `conn`. Assign the returned cursor object to the variable `curs`.

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

curs =
    Attributes: []
    Data: 0
    DatabaseObject: [1x1 database]
    RowLimit: 0
    SQLQuery: 'select * from customers'
    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
```

**Example 2—Select One Column of Data from Database Table**

Select country data from the customers table accessed via the database connection, conn. Assign the SQL statement to the variable sqlquery and assign the returned cursor to the variable curs.

```
sqlquery = 'select country from customers';  
curs = exec(conn, sqlquery);
```

**Example 3—Use Variable in a Query**

Select data from the customers table accessed via the database connection conn, where country is a variable. In this example, the user is prompted to supply their country, which is assigned to the variable UserCountry.

```
UserCountry = input('Enter your country: ', 's')
```

MATLAB prompts

```
Enter your country:
```

The user responds

```
Mexico
```

Without using a variable, the function to retrieve the data would be

```
curs = exec(conn, ['select * from customers where country...  
= 'Mexico'''])  
curs=fetch(curs)
```

To instead perform the query using the user's response, use

```
curs = exec(conn, ['select * from customers where country...  
= '', UserCountry, '''])  
curs=fetch(curs)
```

The select statement is created by using square brackets to concatenate the two strings 'select \* from customers where country =' and 'UserCountry'.

### Example 4—Roll Back or Commit Data Exported to Database Table

Use `exec` to roll back or commit data after running an insert or an update for which the `AutoCommit` flag is off. To roll back data for the database connection `conn`, type

```
exec(conn, 'rollback')
```

To commit the data, type:

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

### Example 5—Run Stored Procedure

Execute the stored procedure `sp_customer_list` for the database connection `conn`.

```
curs = exec(conn, 'sp_customer_list');
```

You can run a stored procedure with input parameters, for example

```
curs = exec(conn, '{call sp_name (parm1,parm2,...)}');
```

### Example 6—Change Catalog

To change the catalog for the database connection `conn` to `intlprice`.

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

### See Also

`close`, `database`, `fetch`, `insert`, `procedures`, `querytimeout`, `resultset`, `rsmd`, `set`, `update`

Notes about database design in “Databases” on page 1-3.

# exportedkeys

---

**Purpose** Get information about exported foreign keys

**Syntax**  
`e = exportedkeys(dbmeta, 'cata', 'sch')`  
`e = exportedkeys(dbmeta, 'cata', 'sch', 'tab')`

**Description** `e = exportedkeys(dbmeta, 'cata', 'sch')` returns the foreign exported key information (that is, information about primary keys that are referenced by other tables), in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`e = exportedkeys(dbmeta, 'cata', 'sch', 'tab')` returns the exported foreign key information (that is, information about the primary key which is referenced by other tables), in the table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

## Examples

Type

```
e = exportedkeys(dbmeta, 'orcl', 'SCOTT')
```

MATLAB returns

```
e =  
Columns 1 through 7  
'orcl' 'SCOTT' 'DEPT' 'DEPTNO' 'orcl' 'SCOTT' 'EMP'  
Columns 8 through 13  
'DEPTNO' '1' 'null' '1' 'FK_DEPTNO' 'PK_DEPT'
```

In this example:

- `dbmeta` is the database metadata object.
- the `cata` field is empty because this database does not include catalogs.
- `SCOTT` is the schema `sch`.

The results show the foreign exported key information.

Column	Description	Value
1	Catalog containing primary key that is exported	null
2	Schema containing primary key that is exported	SCOTT
3	Table containing primary key that is exported	DEPT
4	Column name of primary key that is exported	DEPTNO
5	Catalog that has foreign key	null
6	Schema that has foreign key	SCOTT
7	Table that has foreign key	EMP
8	Foreign key column name, that is the column name that references the primary key in another table	DEPTNO
9	Sequence number within the foreign key	1
10	Update rule, that is, what happens to the foreign key when the primary key is updated.	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, there is only one primary key that is exported to (referenced by) another table. The table DEPT contains a field DEPTNO, its primary key, that 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.

## exportedkeys

---

For a description of the codes for update and delete rules, see <http://java.sun.com/products/jdk/1.2/docs/api/java/sql/package-summary.html> for the DatabaseMetaData object property getExporetedKeys.

### **See Also**

crossreference, dmd, get, importedkeys, primarykeys



---

<b>Purpose</b>	Import data into MATLAB
<b>Syntax</b>	<pre>curs = fetch(curs, RowLimit) curs = fetch(curs) curs.Data</pre>
<b>Description</b>	<p><code>curs = fetch(curs, RowLimit)</code> imports rows of data from the open SQL cursor <code>curs</code>, up to the specified <code>RowLimit</code>, into the object <code>curs</code>. Data is stored in MATLAB in a cell array, structure, or numeric matrix, based on specifications you made using <code>setdbprefs</code>. It is common practice to assign the object returned by <code>fetch</code> to the variable <code>curs</code> from the open SQL cursor. The next time you run <code>fetch</code>, records are imported starting with the row following <code>RowLimit</code>. If you fetch large amounts of data that cause out of memory or speed problems, use <code>RowLimit</code> to limit how much data is retrieved at once.</p> <p><code>curs = fetch(curs)</code> imports rows of data from the open SQL cursor <code>curs</code>, up to the <code>RowLimit</code> specified by <code>set</code>, into the object <code>curs</code>. Data is stored in MATLAB in a cell array, structure, or numeric matrix, based on specifications you made using <code>setdbprefs</code>. It is common practice to assign the object returned by <code>fetch</code> to the variable <code>curs</code> from the open SQL cursor. The next time you run <code>fetch</code>, records are imported starting with the row following <code>RowLimit</code>. If no <code>RowLimit</code> was specified by <code>set</code>, <code>fetch</code> imports all remaining rows of data.</p> <p>Running <code>fetch</code> returns information about the cursor object. The <code>Data</code> element of the cursor object contains the data returned by <code>fetch</code>. The data types are preserved. After running <code>fetch</code>, display the returned data by typing <code>curs.Data</code>.</p> <p>Use <code>get</code> to view properties of <code>curs</code>.</p>
<b>Examples</b>	<p><b>Example 1 – Import All Rows of Data</b></p> <p>Import all of the data into the cursor object <code>curs</code>.</p> <pre>curs = fetch(curs)</pre>

MATLAB returns

```
curs =  
    Attributes: []  
    Data: {91x1 cell}  
    DatabaseObject: [1x1 database]  
    RowLimit: 0  
    SQLQuery: 'select country from customers'  
    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]
```

The fetch operation stores the data in a cell array contained in the cursor object field `curs.Data`. To display data in `curs.Data`, type

```
curs.Data
```

MATLAB returns all of the data, which in this example consists of 1 column and 91 rows, some of which are shown here.

```
ans =  
    'Germany'  
    'Mexico'  
    'Mexico'  
    'UK'  
    'Sweden'  
    ...  
    'USA'  
    'Finland'  
    'Poland'
```

## Example 2—Import Specified Number of Rows of Data

Specify the `RowLimit` argument to retrieve the first 3 rows of data.

```
curs = fetch(curs, 3)
```

MATLAB returns

```
curs =  
    Attributes: []  
        Data: {3x1 cell}  
DatabaseObject: [1x1 database]  
    RowLimit: 0  
    SQLQuery: 'select country from customers'  
    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 data by typing

```
curs.Data
```

MATLAB returns

```
ans =  
    'Germany'  
    'Mexico'  
    'Mexico'
```

Entering the `fetch` function again returns the second 3 rows of data. Adding the semicolon suppresses display of the results.

```
curs = fetch(curs, 3);
```

Display the data by typing

```
curs.Data
```

MATLAB returns

```
ans =  
    'UK'  
    'Sweden'  
    'Germany'
```

## Example 3—Import Numeric Data

Import a column of data that is known to be numeric. Use `setdbprefs` to specify the format for the retrieved data as numeric.

```
conn = database('SampleDB', '', '');
curs=exec(conn, 'select all UnitsInStock from Products');
setdbprefs('DataReturnFormat','numeric')
curs=fetch(curs,3);
curs.Data
```

MATLAB retrieves the data into a numeric matrix.

```
ans =
    39
    17
    13
```

## See Also

`attr`, `cols`, `columnnames`, `exec`, `get`, `rows`, `resultset`, `set`, `width`

**Purpose**

Get object properties

**Syntax**

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

**Description**

`v = get(object)` returns a structure of the properties of `object` and the corresponding property values, assigning the structure to `v`.

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

`v.property` returns the value of `property`, after you have created `v` using `get`.

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

Allowable objects are

- “Database Connection Object”, created using `database`
- “Cursor Object”, created using `exec` or `fetch`
- “Driver Object”, created using `driver`
- “Database Metadata Object”, created using `dmd`
- “Drivermanager Object”, created using `drivermanager`
- “Resultset Object”, created using `resultset`
- “Resultset Metadata Object”, created using `rsmd`

If you are calling these objects from your own Java-based applications, see <http://java.sun.com/products/jdk/1.2/docs/api/java/sql/package-summary.html> for more information about the object properties.

## Database Connection Object

Allowable property names and returned values for a database connection object are listed in the following table.

<b>Property</b>	<b>Value</b>
'AutoCommit'	Status of the AutoCommit flag, either on or off, as specified by set
'Catalog'	Names of catalogs in the data source, for example 'Northwind'
'Driver'	Driver used for the JDBC connection, as specified by database
'Handle'	Identifying JDBC connection object
'Instance'	Name of the data source for an ODBC connection or the 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 a JDBC connection only, the JDBC URL object, jdbc:subprotocol:subname, as specified by database
'UserName'	Username required to connect to the database, as specified by database; note that you cannot use get to retrieve password
'Warnings'	Warnings returned by database

## Cursor Object

Allowable property names and returned values for a cursor object are listed in the following table.

Property	Value
'Attributes'	Cursor attributes
'Data'	Data in the cursor object data element (the query results)
'DatabaseObject'	Information about the database object
'RowLimit'	Maximum number of rows to be returned by fetch, as specified by set
'SQLQuery'	SQL statement for the 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
'Fetch'	0 for cursor created using exec; fetchTheData for cursor created using fetch

## Driver Object

Allowable property names and examples of values for a driver object are listed in the following table.

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

### Database Metadata Object

There are dozens of properties for a database metadata object. Some of the allowable property names and examples of their values are listed 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 Object

Allowable property names and examples of values for a drivermanager object are listed in the following table.

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



## Resultset Object

Some of the allowable property names for a resultset object and examples of their values are listed in the following table.

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

## Resultset Metadata Object

Allowable property names for a resultset metadata object and examples of values are listed 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 the database does 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, Data Source Name

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

```
conn = database('SampleDB', '', '');
v = get(conn, 'Instance')
```

MATLAB returns

```
v =
    SampleDB
```

### Example 2—Get Connection Property, AutoCommit Flag Status

Determine the status of the `AutoCommit` flag for the database connection `conn`.

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

```
ans =
    on
```

### Example 3—Display Data in Cursor

Display the data in the cursor object, `curs` by typing

```
get(curs, 'Data')
```

or by typing

```
curs.Data
```

MATLAB returns

```
ans =  
    'Germany'  
    'Mexico'  
    'France'  
    'Canada'
```

In this example, `curs` contains one column with four records.

#### Example 4—Get Database Metadata Object Properties

View the properties of the database metadata object for connection `conn`. Type

```
dbmeta = dmd(conn);  
v = get(dbmeta)
```

MATLAB returns a list of properties, some of which are shown here.

```
v =  
    AllProceduresAreCallable: 1  
    AllTablesAreSelectable: 1  
    DataDefinitionCausesTransaction: 1  
    DataDefinitionIgnoredInTransact: 0  
    DoesMaxRowSizeIncludeBlobs: 0  
    Catalogs: {4x1 cell}  
    ...  
    NullPlusNonNullIsNull: 0  
    NullsAreSortedAtEnd: 0  
    NullsAreSortedAtStart: 0  
    NullsAreSortedHigh: 0  
    NullsAreSortedLow: 1  
    UsesLocalFilePerTable: 0  
    UsesLocalFiles: 1
```

To view the names of the catalogs in the database, type

```
v.Catalogs
```

# get

---

MATLAB returns the catalog names

```
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, database, dmd, driver, drivermanager, exec, fetch, resultset, rows, rsmd, set

**Purpose** Get information about imported foreign keys

**Syntax**

```
i = importedkeys(dbmeta, 'cata', 'sch')  
i = importedkeys(dbmeta, 'cata', 'sch', 'tab')
```

**Description** `i = importedkeys(dbmeta, 'cata', 'sch')` returns the 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`, where `dbmeta` was created using `dmd`.

`i = importedkeys(dbmeta, 'cata', 'sch', 'tab')` returns the foreign imported key information, that is, information about fields in the table `tab`, that reference primary keys in other tables, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

**Examples**

Type

```
i = importedkeys(dbmeta, 'orcl', 'SCOTT')
```

MATLAB returns

```
i =  
Columns 1 through 7  
'orcl' 'SCOTT' 'DEPT' 'DEPTNO' 'orcl' 'SCOTT' 'EMP'  
Columns 8 through 13  
'DEPTNO' '1' 'null' '1' 'FK_DEPTNO' 'PK_DEPT'
```

In this example:

- `dbmeta` is the database metadata object.
- `orcl` is the catalog `cata`.
- `SCOTT` is the schema `sch`.

# importedkeys

The results show the foreign imported key information as described in the following table.

Column	Description	Value
1	Catalog containing primary key, referenced by foreign imported key	orcl
2	Schema containing primary key, referenced by foreign imported key	SCOTT
3	Table containing primary key, referenced by foreign imported key	DEPT
4	Column name of primary key, referenced by foreign imported key	DEPTNO
5	Catalog that has foreign imported key	orcl
6	Schema that has foreign imported key	SCOTT
7	Table that has foreign imported key	EMP
8	Foreign key column name, that is the column name that references the primary key in another table	DEPTNO
9	Sequence number within foreign key	1
10	Update rule, that is, what happens to the foreign key when the primary key is updated.	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, as well as being the primary key.

For a description of the codes for update and delete rules, see <http://java.sun.com/products/jdk/1.2/docs/api/java/sql/package-summary.html> for the DatabaseMetaData object property getImportedKeys.

### **See Also**

crossreference, dmd, exportedkeys, get, primarykeys

# indexinfo

---

**Purpose** Get indices and statistics for database table

**Syntax** `x = indexinfo(dbmeta, 'cata', 'sch', 'tab')`

**Description** `x = indexinfo(dbmeta, 'cata', 'sch', 'tab')` returns the indices and statistics for the table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

**Examples** Type

```
x = indexinfo(dbmeta, '', 'SCOTT', 'DEPT')
```

MATLAB returns

```
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'
```

In this example:

- `dbmeta` is the database metadata object.
- `orcl` is the catalog `cata`.
- `SCOTT` is the schema `sch`.
- `DEPT` is the table `tab`.

The results contain two rows, meaning there are two index columns. The statistics for the first index column are shown in the following table.



Column	Description	Value
1	Catalog	orcl
2	Schema	SCOTT
3	Table	DEPT
4	Non-unique: 0 if index values can be non-unique, 1 otherwise	0
5	Index catalog	null
6	Index name	null
7	Index type	0
8	Column sequence number within index	0
9	Column name	null
10	Column sort sequence	null
11	Number of rows in the index table or number of unique values in the index	4
12	Number of pages used for the table or number of pages used for the current index	1
13	Filter condition	null

For more information about the index information, see <http://java.sun.com/products/jdk/1.2/docs/api/java/sql/package-summary.html> for a description of the DatabaseMetaData object property `getIndexInfo`.

## See Also

dmd, get, tables

# insert

---

**Purpose** Add MATLAB data to database table

**Syntax** `insert(conn, 'tab', colnames, exdata)`

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

The status of the `AutoCommit` flag determines if `insert` automatically commits the data or if you need to commit the data following the insert. View the `AutoCommit` flag status for the connection using `get` and change it using `set`. Commit the data using `commit` or issue an SQL commit statement via an `exec` function. Roll back the data using `rollback` or issue an SQL rollback statement via an `exec` function.

To replace existing data instead of adding new rows, use `update`.

## Examples

### Example 1 — Insert a Record

Insert one record consisting of two columns, `City` and `Avg_Temp`, into the `Temperatures` table. The data is San Diego, 88 degrees. The database connection is `conn`.

Assign the data to the cell array.

```
exdata = {'San Diego', 88}
```

Create a cell array containing the column names in `Temperatures`.

```
colnames = {'City', 'Avg_Temp' }
```

Perform the insert.

```
insert(conn, 'Temperatures', colnames, exdata)
```

The row of data is added to the `Temperatures` table.

### Example 2—Insert Multiple Records

Insert a cell array, `exdata`, containing 28 rows of data with three columns, into the `Growth` table. The data columns are `Date`, `Avg_Length`, and `Avg_Wt`. The database connection is `conn`.

Insert the data.

```
insert(conn, 'Growth', {'Date'; 'Avg_Length'; 'Avg_Wt'}, exdata)
```

The records are inserted in the table.

### Example 3—Import Records, Perform Computations, and Export Data

Perform calculations on imported data and then export the data. First import all of the data in the `products` table. Because the data contains numeric and character data, import the data into a cell array.

```
conn = database('SampleDB', '', '');  
curs = exec(conn, 'select * from products');  
setdbprefs('DataReturnFormat', 'cellarray')  
curs = fetch(curs);
```

Assign the first column of data to the variable `id`.

```
id = curs.Data(:,1);
```

Assign the sixth column of data to the variable `price`.

```
price = curs.Data(:,6);
```

Calculate the discounted price (25% off) and assign it to the variable `new_price`. You must convert the cell array `price` to a numeric matrix in order to perform the calculation.

```
new_price = .75*[price{:}]
```

Export the `id`, `price`, and `new_price` data to the `Sale` table. Because `id` is a character array and `new_price` is numeric, put the exported data in a cell array. The variable `new_price` is a numeric matrix because it was the result of the discount calculation. You must convert `new_price` to a cell array. To convert the columns of data in `new_price` to a cell arrays, type

```
new_price = num2cell(new_price);
```

# insert

---

Create an array, `exdata`, that contains the three columns of data to be exported. Put the `id` data in column one, `price` in column two, and `new_price` in column three.

```
exdata(:,1) = id(:,1);  
exdata(:,2) = price;  
exdata(:,3) = new_price;
```

Assign the column names to a string array, `colnames`.

```
colnames={'product_id', 'price', 'sale_price'};
```

Export the data to the `Sale` table.

```
insert(conn, 'Sale', colnames, exdata)
```

All rows of data are inserted into the `Sale` table.

## Example 4—Insert Numeric Data

Export the `new_price` data into the `sale_price` column of the `Sale` table, where `new_price` is a numeric matrix.

```
insert(conn, 'Sale', {'sale_price'}, new_price)
```

When exporting, you do not need to define the type of data you are exporting.

## Example 5—Insert Followed by commit

This example demonstrates the use of the SQL `commit` function following an `insert`. The `AutoCommit` flag is off.

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

```
insert(conn, 'Error_Rate', colnames, exdata)
```

Commit the data using the `commit` function.

```
commit(conn)
```

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

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

**See Also**

commit, database, exec, rollback, set, update

# isconnection

---

**Purpose** Detect if database connection is valid

**Syntax** `a = isconnection(conn)`

**Description** `a = isconnection(conn)` returns 1 if the database connection `conn` is valid, or returns 0 otherwise, where `conn` was created using `database`.

**Examples** Type  
`a = isconnection(conn)`  
and MATLAB returns  
`a =`  
`1`  
indicating that the database connection `conn` is valid.

**See Also** `database`, `isreadonly`, `ping`

- Purpose** Detect if driver is a valid JDBC driver object
- Syntax** `a = isdriver(d)`
- Description** `a = isdriver(d)` returns 1 if `d` is a valid JDBC driver object, or returns 0 otherwise, where `d` was created using `driver`.
- Examples** Type  
`a = isdriver(d)`  
and MATLAB returns  
`a =`  
    1  
indicating that the database driver object `d` is valid.
- See Also** `driver`, `get`, `isjdbc`, `isurl`

# isjdbc

---

**Purpose** Detect if driver is JDBC-compliant

**Syntax** `a = isjdbc(d)`

**Description** `a = isjdbc(d)` returns 1 if the driver object `d` is JDBC compliant, or returns 0 otherwise, where `d` was created using `driver`.

**Examples** Type

```
a = isjdbc(d)
```

and MATLAB returns

```
a =  
    1
```

indicating that the database driver object `d` is JDBC compliant.

**See Also** `driver`, `get`, `isdriver`, `isurl`



**Purpose** Detect if last record read in resultset was NULL

**Syntax** `a = isnullcolumn(rset)`

**Description** `a = isnullcolumn(rset)` returns 1 if the last record read in the resultset `rset`, was NULL, and returns 0 otherwise.

**Examples** **Example 1—Result Is Not NULL**

Type

```
curs = fetch(curs,1);
rset = resultset(curs);
isnullcolumn(rset)
```

MATLAB returns

```
ans =
     0
```

indicating that the last record of data retrieved was *not* NULL. To verify this, type

```
curs.Data
```

MATLAB returns

```
ans =
 [1400]
```

**Example 2—Result Is NULL**

```
curs = fetch(curs,1);
rset = resultset(curs);
isnullcolumn(rset)
```

MATLAB returns

```
ans =
     1
```

indicating that the last record of data retrieved was NULL. To verify this, type

```
curs.Data
```

# isnullcolumn

---

MATLAB returns

```
ans =  
    [NaN]
```

## See Also

get, resultset

**Purpose** Detect if database connection is read only

**Syntax** `a = isreadonly(conn)`

**Description** `a = isreadonly(conn)` returns 1 if the database connection `conn` is read only, or returns 0 otherwise, where `conn` was created using `database`.

**Examples** Type  
`a = isreadonly(conn)`

and MATLAB returns

```
a =  
    1
```

indicating that the database connection `conn` is read only. Therefore, you cannot perform `insert` or `update` functions for this database.

**See Also** `database`, `isconnection`

# isurl

---

**Purpose** Detect if the database URL is valid

**Syntax** `a = isurl('s', d)`

**Description** `a = isurl('s', d)` returns 1 if the database URL `s`, for the driver object `d`, is valid, or returns 0 otherwise. The URL `s` is of the form `jdbc:odbc:<name>` or `<name>`, and `d` is the driver object created using `driver`.

**Examples** Type

```
a = isurl('jdbc:odbc:thin:@144.212.123.24:1822:', d)
```

and MATLAB returns

```
a =  
    1
```

indicating that the database URL, `jdbc:odbc:thin:@144.212.123.24:1822:`, is valid for driver object `d`.

**See Also** `driver`, `get`, `isdriver`, `isjdbc`

**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, allowed for a MATLAB session to try to connect to a database via the specified JDBC driver. Use `logintimeout` before running the database function. If MATLAB cannot connect within the allowed 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 MATLAB cannot connect within the allowed time, it stops trying.

`timeout = logintimeout('driver')` returns the time, in seconds, you set previously using `logintimeout` for the JDBC connection specified by `driver`. A returned value of zero means that the time-out value has not been set previously; MATLAB stops trying to make a connection if it is not immediately successful.

`timeout = logintimeout` returns the time, in seconds, you set previously using `logintimeout` for an ODBC connection. A returned value of zero means that the time-out value has not been set previously; MATLAB stops trying to make a connection if it is not immediately successful.

If you do not use `logintimeout` and MATLAB tries to connect without success, your MATLAB session could hang up.

# logintimeout

---

## Examples

### Example 1—Get Time-out Value for ODBC Connection

Your database connection is via an ODBC connection. To see the current time-out value, type

```
logintimeout
```

MATLAB returns

```
ans =  
    0
```

The time-out value has not been set.

### Example 2—Set Time-out Value for ODBC Connection

Set the time-out value to 5 seconds for an ODBC driver. Type

```
logintimeout(5)
```

MATLAB returns

```
ans =  
    5
```

### Example 3—Get and Set Time-out Value for JDBC Connection

Your database connection is via the Oracle JDBC driver. First see what the current time-out value is. Type

```
logintimeout('oracle.jdbc.driver.OracleDriver')
```

MATLAB returns

```
ans =  
    0
```

The time-out value is currently 0. Set the time-out to 10 seconds. Type

```
timeout = logintimeout('oracle.jdbc.driver.OracleDriver', 10)
```

MATLAB returns

```
timeout =  
    10
```

Verify the time-out value for the JDBC driver. Type

```
logintimeout('oracle.jdbc.driver.OracleDriver')
```

MATLAB returns

```
ans =  
    10
```

## See Also

database, get, set

# namecolumn

---

**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, for the resultset `rset`, where `rset` was created using `resultset`, and `n` is a string or cell array of strings containing the column names. Get the column names for a given cursor using `columnnames`.

**Examples** Type

```
x = namecolumn(rset, {'DNAME';'LOC'})
```

MATLAB returns

```
x =  
    2    3
```

In this example, the resultset object is `rset`. The column names for which you want the column index are `DNAME` and `LOC`. The results show that `DNAME` is column 2 and `LOC` is column 3.

To get the index for only the `LOC` column, type

```
x = namecolumn(rset, 'LOC')
```

**See Also** `columnnames`, `resultset`



- Purpose** Get status information about database connection
- Syntax** `ping(conn)`
- Description** `ping(conn)` returns the status information about the database connection, `conn`. If the connection is open, `ping` returns status information and otherwise it returns an error message.

**Examples** **Example 1 – Get Status Information About ODBC Connection**

Type

```
ping(conn)
```

where `conn` is a valid ODBC connection. MATLAB returns

```
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:SampleDB'  
    AutoCommitTransactions: 'True'
```

## Example 2—Get Status Information About JDBC Connection

Type

```
ping(conn)
```

where conn is a valid JDBC connection.

MATLAB returns

```
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

Type

```
ping(conn)
```

where the database connection conn has been terminated or was not successful.

MATLAB returns

```
Cannot Ping the Database Connection
```

### See Also

database, dmd, get, isconnection, set, supports

**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 the primary key information for all tables in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`k = primarykeys(dbmeta, 'cata', 'sch', 'tab')` returns the primary key information for the table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

**Examples**

Type

```
k = primarykeys(dbmeta, 'orcl', 'SCOTT', 'DEPT')
```

MATLAB returns

```
k =  
    'orcl'    'SCOTT'    'DEPT'    'DEPTNO'    '1'    'PK_DEPT'
```

In this example:

- `dbmeta` is the database metadata object.
- `orcl` is the catalog `cata`.
- `SCOTT` is the schema `sch`.
- `DEPT` is the table `tab`.

# primarykeys

---

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

**Purpose** Get catalog's stored procedure parameters and result columns

**Syntax**

```
pc = procedurecolumns(dbmeta, 'cata')  
pc = procedurecolumns(dbmeta, 'cata', 'sch')
```

**Description** `pc = procedurecolumns(dbmeta, 'cata')` returns the stored procedure parameters and result columns for the catalog `cata`, for the database whose database metadata object is `dbmeta`, which was created using `dmd`.

`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`, which was created using `dmd`.

MATLAB returns one row for each column in the results generated by running the stored procedure.

**Examples**

```
Type  
pc = procedurecolumns(dbmeta, 'tutorial', 'ORG')
```

where:

- `dbmeta` is the database metadata object.
- `tutorial` is the catalog `cata`.
- `ORG` is the schema `sch`.

MATLAB returns

```
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'
```

# procedurecolumns

---

The results show the stored procedure parameter and result information. Because two rows of data are returned, there will be two columns of data in the results when you run the stored procedure. From the results, you can see that running the stored procedure `display` returns the Month and Day. 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 <http://java.sun.com/products/jdk/1.2/docs/api/java/sql/package-summary.html> for the `DatabaseMetaData` object property `getProcedureColumns`.

## See Also

`dmd`, `get`, `procedures`

**Purpose** Get catalog's stored procedures

**Syntax**

```
p = procedures(dbmeta, 'cata')  
p = procedures(dbmeta, 'cata', 'sch')
```

**Description** `p = procedures(dbmeta, 'cata')` returns the stored procedures in the catalog `cata`, for the database whose database metadata object is `dbmeta`, which was created using `dmd`.

`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`, which was created using `dmd`.

Stored procedures are SQL statements that are saved with the database. You can use the `exec` function to run a stored procedure, providing the stored procedure as the `sqlquery` argument instead of actually entering the `sqlquery` statement as the argument.

**Examples** Type

```
p = procedures(dbmeta, 'DBA')
```

where `dbmeta` is the database metadata object and the catalog is `DBA`. MATLAB returns the names of the stored procedures

```
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 of the data. Type

```
curs = exec(conn, 'sp_customer_list');  
curs = fetch(conn)
```

MATLAB returns

```
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 by typing

```
curs.Data
```

MATLAB returns

```
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

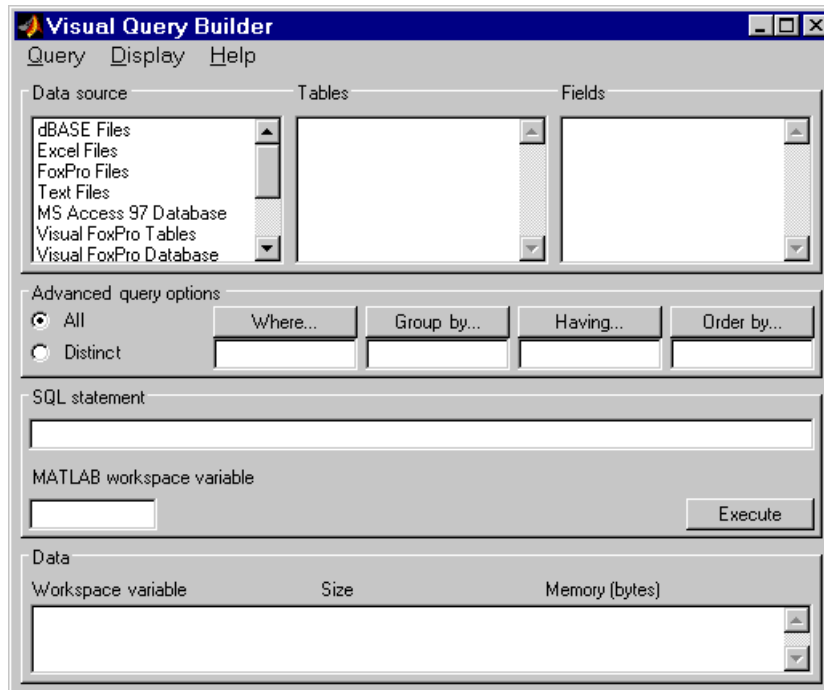
dmd, exec, get, procedurecolumns



**Purpose** Start visual SQL query builder

**Syntax** querybuilder

**Description** querybuilder starts the Visual Query Builder (VQB), an easy to use interface for building and running SQL queries to retrieve data from databases.



**Examples** For examples of and more information about using the Visual Query Builder, use the VQB **Help** menu or see “Visual Query Builder” on page 2-1. You can also get help in any of the Visual Query Builder dialog boxes by clicking the **Help** button in the dialog box.

# querytimeout

---

**Purpose** Get time allowed for a database SQL query to succeed

**Syntax** `timeout = querytimeout(curs)`

**Description** `timeout = querytimeout(curs)` returns the amount of time, in seconds, allowed for an SQL query of `curs` to succeed, where `curs` is created by running `exec`. If a query cannot be completed in the allowed time, MATLAB stops trying to perform the `exec`. The time-out value is defined for a database by the database administrator. If the time-out value is zero, a query must be completed immediately.

**Examples** Get the current database time-out setting for `curs`.

```
querytimeout(curs)
ans =
    10
```

**Limitations** If a database does not have a database time-out feature, MATLAB returns  
`[Driver]Driver not capable`

The Microsoft Access ODBC driver and Oracle ODBC driver do not support `querytimeout`.

**See Also** `exec`

**Purpose** Load database driver

**Syntax** `register(d)`

**Description** `register(d)` loads the database driver object `d`, which was created using `driver`. Use `unregister` to unload the driver.

Although database automatically loads the driver, `register` allows you to get properties of the driver before connecting. The `register` function also allows you to use `drivermanager` to set and get properties for all loaded drivers.

**Examples** `register(d)` loads the database driver object `d`.

`get(d)` returns properties of the driver object.

**See Also** `driver`, `drivermanager`, `get`, `unregister`

# resultset

---

**Purpose** Construct resultset object

**Syntax** `rset = resultset(curs)`

**Description** `r = resultset(curs)` creates a resultset object `rset`, for the cursor `curs`, where `curs` was created using `exec` or `fetch`. You can get properties of `rset`, create a resultset metadata object using `rsmd`, or make calls to `rset` using your own Java-based applications. You can also perform other functions on `rset`: `clearwarnings`, `isnullcolumn`, and `namecolumn`. Use `close` to close the resultset, which frees up resources.

**Examples** Type  
`rset = resultset(curs)`

MATLAB returns

```
rset =  
    Handle: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
```

**See Also** `clearwarnings`, `close`, `exec`, `fetch`, `get`, `isnullcolumn`, `namecolumn`, `rsmd`

<b>Purpose</b>	Undo database changes
<b>Syntax</b>	<code>rollback(conn)</code>
<b>Description</b>	<code>rollback(conn)</code> reverses changes made via <code>insert</code> or <code>update</code> to the database connection <code>conn</code> . The <code>rollback</code> function reverses all changes made since the last <code>commit</code> or <code>rollback</code> , or the last <code>exec</code> that performed a <code>commit</code> or <code>rollback</code> . The <code>AutoCommit</code> flag for <code>conn</code> must be <code>off</code> to use <code>rollback</code> .
<b>Examples</b>	<p>Ensure the <code>AutoCommit</code> flag for connection <code>conn</code> is <code>off</code> by typing</p> <pre>get(conn, 'AutoCommit')</pre> <p>MATLAB returns</p> <pre>ans =     off</pre> <p>Insert the data contained in <code>exdata</code> into the columns <code>DEPTNO</code>, <code>DNAME</code>, and <code>LOC</code>, in the table <code>DEPT</code>, for the data source <code>conn</code>. Type</p> <pre>insert(conn, 'DEPT', {'DEPTNO'; 'DNAME'; 'LOC'}, exdata)</pre> <p>Roll back the data inserted in the database by typing</p> <pre>rollback(conn)</pre> <p>The data in <code>exdata</code> is removed from the database so the database contains the same data it did before the <code>insert</code>.</p>
<b>See Also</b>	<code>commit</code> , <code>database</code> , <code>exec</code> , <code>get</code> , <code>insert</code> , <code>update</code>

# rows

---

**Purpose** Get number of rows in fetched data set

**Syntax** `numrows = rows(curs)`

**Description** `numrows = rows(curs)` returns the number of rows in the fetched data set `curs`.

**Examples** There are four rows in the fetched data set `curs`.

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

```
numrows =  
    4
```

To see the four rows of data in `curs`, type

```
curs.Data
```

MATLAB returns

```
ans =  
    'Germany'  
    'Mexico'  
    'France'  
    'Canada'
```

**See Also** `cols`, `fetch`, `get`, `rsmd`

---

<b>Purpose</b>	Construct resultset metadata object
<b>Syntax</b>	<pre>rsmeta = rsmd(rset) rsmeta = rsmd(curs)</pre>
<b>Description</b>	<p><code>rsmeta = rsmd(rset)</code> creates a resultset metadata object <code>rsmeta</code>, for the resultset object <code>rset</code>, or the cursor object <code>curs</code>, where <code>rset</code> was created using <code>resultset</code>, and <code>curs</code> was created using <code>exec</code> or <code>fetch</code>. Get properties of <code>rsmeta</code> using <code>get</code>, or make calls to <code>rsmeta</code> using your own Java-based applications.</p>
<b>Examples</b>	<p>Type</p> <pre>rsmeta=rsmd(rset)</pre> <p>MATLAB returns</p> <pre>rsmeta =     Handle: [1x1 sun.jdbc.odbc.JdbcOdbcResultSetMetaData]</pre> <p>Use <code>v = get(rsmeta)</code> and <code>v.property</code> to see properties of the resultset metadata object.</p>
<b>See Also</b>	<code>exec</code> , <code>get</code> , <code>resultset</code>

# set

---

**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 you can set for object are

- “Database Connection Object”, created using database
- “Cursor Object”, created using exec or fetch
- “Drivermanager Object”, created using drivermanager

Not all databases allow you to set all of these properties. If your database does not allow you to set a particular property, you will receive an error message when you try to do so.



## Database Connection Object

The allowable values for property and value for a database connection object are listed in the following table.

Property	Value	Description
'AutoCommit'	'on'	Database data is written and committed automatically when you run an insert or update function. You cannot use rollback to reverse it and you do not need to use commit because the data is committed automatically.
	'off'	Database data is not committed automatically when you run an insert or update function. In this case, after you run insert or update, you can use rollback to reverse the insert or update. When you are sure the data is correct, follow an insert or update with a commit.
'ReadOnly'	0	<i>Not</i> read-only, that is, writable
	1	Read-only
'TransactionIsolation'	positive integer	Current transaction isolation level

Note that if you do not run commit after running an update or insert function, and then close the database connection using close, the data usually is committed automatically at that time. Your database administrator can tell you how your database deals with this.

# set

## Cursor Object

The allowable property and value for a cursor object are listed in the following table.

Property	Value	Description
'RowLimit'	positive integer	Sets the RowLimit for fetch. This is an alternative to defining the RowLimit as an argument of fetch. Note that the behavior of fetch when you define RowLimit using set differs depending on the database.

## Drivermanager Object

The allowable property and value for a drivermanager object are listed in the following table.

Property	Value	Description
'LoginTimeout'	positive integer	Sets the logintimeout value for the set of loaded database drivers as a whole.

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 uses set to define the RowLimit. It establishes a JDBC connection, retrieves all data from the EMP table, sets the RowLimit to 5, and uses fetch with no arguments to retrieve the data.

Only five rows of data are returned by fetch.

```
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]
```

As seen above, the RowLimit property of curs is now 5 and the Data property is 5x8 cell, meaning five rows of data were returned.

For the database in this example, the RowLimit acts as the maximum number of rows you can retrieve. Therefore, if you run the fetch function again, no data is returned.

### Example 2—Set AutoCommit Flag to On for Connection

This example shows a database update when the AutoCommit flag is on. First determine the status of the AutoCommit flag for the database connection conn.

```
get(conn, 'AutoCommit')

ans =
    off
```

The flag is off.

Set the flag status to on and verify it.

```
set(conn, 'AutoCommit', 'on');
get(conn, 'AutoCommit')

ans =
on
```

Insert data, cell array `exdata`, into the column names `colnames`, of the `Growth` table.

```
insert(conn, 'Growth', colnames, exdata)
```

The data is inserted and committed.

### **Example 3—Set AutoCommit Flag to Off for Connection and Commit Data**

This example shows a database insert when the `AutoCommit` flag is off and the data is then committed. First set the `AutoCommit` flag to off for database connection `conn`.

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

Insert data, cell array `exdata`, into the column names `colnames`, of the `Avg_Freight_Cost` table.

```
insert(conn, 'Avg_Freight_Cost', colnames, exdata)
```

Commit the data.

```
commit(conn)
```

### **Example 4—Set AutoCommit Flag to Off for Connection and Roll Back Data**

This example shows a database update when the `AutoCommit` flag is off and the data is then rolled back. First set the `AutoCommit` flag to off for database connection `conn`.

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

Update the data in the column names specified by `colnames`, of the `Avg_Freight_Weight` table, for the record selected by `whereclause`, using data contained in cell array `exdata`.

```
update(conn, 'Avg_Freight_Weight', colnames, exdata, whereclause)
```

The data was written but not committed.

Roll back the data.

```
rollback(conn)
```

The data in the table is now the same as it was before update was run.

### Example 5—Set LoginTimeout for Drivermanager Object

In this example, create a `drivermanager` object `dm`, and set the `LoginTimeout` value to 3 seconds. Type:

```
dm = drivermanager;  
set(dm, 'LoginTimeout', 3);
```

To verify the result, type

```
logintimeout
```

MATLAB returns

```
ans =  
    3
```

### See Also

`database`, `drivermanager`, `exec`, `fetch`, `get`, `insert`, `logintimeout`, `ping`, `update`

# setdbprefs

---

<b>Purpose</b>	Set preferences for data retrieval format, error notification, and NULL values
<b>Graphical Interface</b>	As an alternative to the setdbprefs function, you can select <b>Preferences</b> from the Visual Query Builder <b>File</b> menu and use the <b>Preferences</b> dialog box.
<b>Syntax</b>	<pre>setdbprefs setdbprefs('property') setdbprefs('property', 'value') setdbprefs({'property1'; ... }, {'value1'; ... })</pre>
<b>Description</b>	<p>setdbprefs returns the current values for database action preferences.</p> <p>setdbprefs('property') returns the current preference value for the specified property.</p> <p>setdbprefs('property', 'value') sets the preference to value for the specified property for the current session.</p> <p>setdbprefs({'property1'; ... }, {'value1'; ... }) for the properties starting with property1, sets the preference values starting with value1, for the current session.</p>

Allowable properties are listed in the following table.

<b>Allowable Properties</b>	<b>Allowable Values</b>	<b>Description</b>
'DataReturnFormat'		Format for data imported into MATLAB. Select a value based on the type of data you are importing, memory considerations, and your preferred method of working with retrieved data. Set the value before using <code>fetch</code> .
	'cellarray' (default)	Imports data into MATLAB cell arrays. Use for non-numeric data types. Requires substantial system memory when retrieving large amounts of data. Has slower performance than numeric format. To address memory problems, use the <code>RowLimit</code> option with <code>fetch</code> . For more information about cell arrays, see “Working with Cell Arrays in MATLAB” on page 3-36.
	'numeric'	Imports data into a MATLAB matrix of doubles. Non-numeric data types are considered to be NULL numbers and are shown as specified for the <code>NullNumberRead</code> property. Uses less system memory and offers better performance than the <code>cellarray</code> format. Use only when data to be retrieved is in numeric format, or when the nonnumeric data retrieved is not relevant.
	'structure'	Imports data as a MATLAB structure. Can use for all data types. Makes it easy to work with returned columns. Requires substantial system memory when retrieving large amounts of data. Has slower performance than numeric format. To address memory problems, use the <code>RowLimit</code> option with <code>fetch</code> . For more information on using structures, see “Structures and Cell Arrays” in the MATLAB documentation.

# setdbprefs

Allowable Properties	Allowable Values	Description (Continued)
'ErrorHandling'		Behavior for handling errors when importing data. Set the value before running <code>exec</code> .
	'store' (default)	Any errors from running database are stored in the <code>Message</code> field of the returned connection object. Any errors from running <code>exec</code> are stored in the <code>Message</code> field of the returned cursor object.
	'report'	Any errors from running database or <code>exec</code> display immediately in the <b>Command Window</b> .
	'empty'	Any errors from running database are stored in the <code>Message</code> field of the returned connection object. Any 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> .
'NullNumberRead'	User-specified, for example, '0'	How NULL numbers in a database are represented when imported into MATLAB. NaN is the default value. Cannot specify a string value, such as 'NULL', if 'DataReturnFormat' is set to 'numeric'. Set the value before using <code>fetch</code> .
'NullNumberWrite'	User-specified, for example, 'NaN'	How NULL numbers in MATLAB are represented when exported to a database. NaN is the default value.
'NullStringRead'	User-specified, for example, 'null'	How NULL strings in a database are represented when imported into MATLAB. NaN is the default value. Set the value before using <code>fetch</code> .
'NullStringWrite'	User-specified, for example, 'NULL'	How NULL strings in MATLAB are represented when exported to a database. NaN is the default value.



## Examples

### Example 1 – Display Current Values

Type `setdbprefs` and MATLAB returns

```
DataReturnFormat: 'cellarray'  
ErrorHandling: 'store'  
NullNumberRead: 'NaN'  
NullNumberWrite: 'NULL'  
NullStringRead: 'null'  
NullStringWrite: 'null'
```

which means

- Data is imported into MATLAB cell arrays.
- Any errors that occur during a connection or a SQL query are stored in the Message field of the connection or cursor data object.
- Any NULL number in the database is read into MATLAB as NaN. Any NaN number in MATLAB is exported to the database as a NULL number. Any NULL string in the database is read into MATLAB as 'null'. Any 'null' string in MATLAB is exported to the database as a NULL string.

### Example 2 – Change a Value

Type `setdbprefs ('NullNumberRead')` and MATLAB returns

```
NullNumberRead: '0'
```

which means that any NULL number in the database is read into MATLAB as NaN.

To change the value to 0, type

```
setdbprefs ('NullNumberRead', '0')
```

which means that any NULL number in the database is read into MATLAB as 0.

## Example 3—Change the DataReturnFormat

**Cell array:** To specify the cellarray format, type

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

which means that data is imported into MATLAB cell arrays. The following illustrates a subsequent import.

```
conn = database('SampleDB', '', '');
curs=exec(conn, 'select all ProductName,UnitsInStock from...
Products');
curs=fetch(curs,3);
curs.Data
ans =
    'Chai'           [39]
    'Chang'          [17]
    'Aniseed Syrup' [13]
```

**Numeric:** Specify the numeric format by typing

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

Performing the same set of import functions results in

```
curs.Data
ans =
    NaN    39
    NaN    17
    NaN    13
```

In the database, the values for ProductName are all character strings, as seen in the previous results when DataReturnFormat is set to cellarray. The ProductName values cannot be read when they are imported using the numeric format. Therefore, MATLAB treats them as NULL numbers and assigns them as NaN, which is the current value for the NullNumberRead property of setdbprefs in this example.

**Structure:** Specify the structure format by typing

```
setdbprefs ('DataReturnFormat','structure')
```

Performing the same set of import functions results in

```
curs.Data
ans =
    ProductName: {3x1 cell}
    UnitsInStock: [3x1 double]
```

View the contents of the structure to see the data.

```
curs.Data.ProductName
ans =
    'Chai'
    'Chang'
    'Aniseed Syrup'

curs.Data.UnitsInStock
ans =
    39
    17
    13
```

## Example 4—Change the ErrorHandling

**Store:** To specify the store format, type

```
setdbprefs ('ErrorHandling','store')
```

which means that any errors from running database or exec are stored in the Message field of the returned connection or cursor object.

The following illustrates an example of trying to fetch from a closed cursor.

```
conn = database('SampleDB', '', '');
curs=exec(conn, 'select all ProductName from Products');
close(curs)
curs=fetch(curs,3);
curs=

    Attributes: []
           Data: 0
 DatabaseObject: [1x1 database]
           RowLimit: 0
           SQLQuery: 'select all ProductName from Products'
           Message: 'Error: Invalid cursor'
           Type: 'Database Cursor Object'
           ResultSet: 0
           Cursor: 0
           Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
           Fetch: [1x1 ...
                  com.mathworks.toolbox.database.fetchTheData]
```

The error indication appears in the Message field.

**Report:** To specify the report format, type

```
setdbprefs ('ErrorHandling','report')
```

which means that any errors from running database or exec display immediately in the **Command Window**.

The following illustrates the same example as above when trying to fetch from a closed cursor.

```
conn = database('SampleDB', '', '');
curs=exec(conn, 'select all ProductName from Products');
close(curs)
curs=fetch(curs,3);
?? Error using ==> cursor/fetch (errorhandling)

Invalid Cursor

Error in ==> D:\matlab\toolbox\database\database\@cursor\fetch.m
On line 36 ==> errorhandling(initialCursor.Message);
```

The error indication appears immediately in the **Command Window**.

**Empty:** To specify the empty format, type

```
setdbprefs ('ErrorHandling','empty')
```

which means that any errors from 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, [].

The following illustrates the same example as above when trying to fetch from a closed cursor.

```
conn = database('SampleDB', '', '');
curs=exec(conn, 'select all ProductName from Products');
close(curs)
curs=fetch(curs,3);
curs =

    Attributes: []
           Data: []
DatabaseObject: [1x1 database]
      RowLimit: 0
      SQLQuery: 'select all ProductName from Products'
      Message: 'Invalid Cursor'
           Type: 'Database Cursor Object'
      ResultSet: 0
           Cursor: 0
      Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
           Fetch: [1x1 ...
                  com.mathworks.toolbox.database.fetchTheData]
```

The error indication appears in the cursor object Message field. In addition, the Attributes field returned empty handles since no attributes could be created.

## Example 5—Change Multiple Settings

Type

```
setdbprefs({'NullStringRead'; 'DataReturnFormat'}, ...
{'NaN'; 'numeric'})
```

which means any NULL string in the database is read into MATLAB as 'NaN', and data is retrieved into a matrix of doubles.

**Purpose** Convert JDBC SQL grammar to system's native SQL grammar

**Syntax** `n = sql2native(conn, 'sqlquery')`

**Description** `n = sql2native(conn, 'sqlquery')` for the connection `conn`, which was created using `database`, converts the SQL statement string `sqlquery`. The string is converted from JDBC SQL grammar into the database system's native SQL grammar, returning the native SQL statement to `n`.

# supports

---

## Purpose

Detect if property is supported by database metadata object

## Syntax

```
a = supports(dbmeta)
a = supports(dbmeta, 'property')
a.property
```

## Description

`a = supports(dbmeta)` returns a structure of the properties of `dbmeta`, which was created using `dmd`, and the corresponding property values, 1 or 0, where 1 means the property is supported and 0 means the property is not supported.

`a = supports(dbmeta, 'property')` returns the value, 1 or 0, of `property` for `dbmeta`, which was created using `dmd`, where 1 means the property is supported and 0 means the property is not supported.

`a.property` returns the value of `property`, after you created `a` using `supports`.

There are dozens of properties for `dbmeta`. Examples include 'GroupBy' and 'StoredProcedures'.

## Examples

Type

```
a = supports(dbmeta, 'GroupBy')
```

and MATLAB returns

```
a =
    1
```

indicating that the database supports the use of SQL group-by clauses.

To find the GroupBy value as well as values for all other properties, type

```
a = supports(dbmeta)
```

MATLAB returns a list of properties and their values. The GroupBy property is included in the list. You can also see its value by typing

```
a.GroupBy
```

to which MATLAB returns

```
a =
    1
```



**See Also**

database, dmd, get, ping

# tableprivileges

---

**Purpose** Get 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 the list of table privileges for all tables in the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`tp = tableprivileges(dbmeta, 'cata', 'sch')` returns the list of table privileges for all tables in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`tp = tableprivileges(dbmeta, 'cata', 'sch', 'tab')` returns the list of privileges for the table `tab`, in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

**Examples** Type

```
tp = tableprivileges(dbmeta, 'msdb', 'geck', 'builds')
```

MATLAB returns

```
tp =
    'DELETE'    'INSERT'    'REFERENCES'    'SELECT'    'UPDATE'
```

In this example:

- `dbmeta` is the database metadata object.
- `msdb` is the catalog `cata`.
- `geck` is the schema `sch`.
- `builds` is the table `tab`.

The results show the set of privileges.

**See Also** `dmd`, `get`, `tables`

**Purpose** Get database table names

**Syntax**

```
t = tables(dbmeta, 'cata')
t = tables(dbmeta, 'cata', 'sch')
```

**Description** `t = tables(dbmeta, 'cata')` returns the list of all tables and their table types in the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`t = tables(dbmeta, 'cata', 'sch')` returns the list of tables and table types in the schema `sch`, of the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

For command line help on `tables`, use the overloaded method

```
help dmd/tables
```

## Examples

Type

```
t = tables(dbmeta, 'orcl', 'SCOTT')
```

MATLAB returns

```
t =
      'BONUS'      'TABLE'
      'DEPT'      'TABLE'
      'EMP'        'TABLE'
      'SALGRADE'  'TABLE'
      'TRIAL'     'TABLE'
```

In this example:

- `dbmeta` is the database metadata object.
- `orcl` is the catalog `cata`.
- `SCOTT` is the schema `sch`.

The results show the names and types of the five tables.

## See Also

`attr`, `bestrowid`, `dmd`, `get`, `indexinfo`, `tableprivileges`

# unregister

---

**Purpose** Unload database driver

**Syntax** `unregister(d)`

**Description** `unregister(d)` unloads the database driver object `d`, which was loaded using `register`. Running `unregister` frees up system resources. If you do not use `unregister` to unload a registered driver, it automatically unloads when you end the MATLAB session.

**Examples** `unregister(d)` unloads the database driver object `d`.

**See Also** `register`

**Purpose** Replace data in database table with data from MATLAB

**Syntax** `update(conn, 'tab', colnames, exdata, 'whereclause')`

**Description** `update(conn, 'tab', colnames, exdata, 'whereclause')` exports data from the MATLAB variable `exdata`, into the database table `tab`, via the database connection `conn`. The variable `exdata` can be a cell array, numeric matrix, or structure. You do not define the type of data you are exporting; the data is exported in its current MATLAB format. Existing records in the table are replaced as specified by the SQL command `whereclause`. Specify the column names for `tab` as strings in the MATLAB cell array, `colnames`. If `exdata` is a structure, field names in the structure must exactly match `colnames`.

The status of the `AutoCommit` flag determines if `update` automatically commits the data or if a `commit` is needed. View the `AutoCommit` flag status for the connection using `get` and change it using `set`. Commit the data using `commit` or issue an SQL `commit` statement via the `exec` function. Roll back the data using `rollback` or issue an SQL `rollback` statement via the `exec` function.

To add new rows instead of replacing existing data, use `insert`.

## Examples

### Example 1—Update a Record

In the `Birthdays` table, update the record where `First_Name` is `Jean`, replacing the current value for `Age` with the new value, `40`. The connection is `conn`.

Define a cell array containing the column name you are updating, `Age`.

```
colnames = {'Age'}
```

Define a cell array containing the new data.

```
exdata(1,1) = {40}
```

Perform the update.

```
update(conn, 'Birthdays', colnames, exdata, ...  
       'where First_Name = ''Jean''')
```

## Example 2—Update Followed by rollback

This example shows a database update when the AutoCommit flag is off and the data is then rolled back. First set the AutoCommit flag to off for database connection conn.

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

Update the data in the column Date of the Error\_Rate table for the record selected by whereclause using data contained in the cell array exdata.

```
update(conn, 'Error_Rate', {'Date'}, exdata, whereclause)
```

The data was written, but not committed.

Roll back the data.

```
rollback(conn)
```

The update was reversed; the data in the table is the same as it was before update was run.

## See Also

commit, database, insert, rollback, set

## Purpose

Get automatically updated table columns

## Syntax

```
v1 = versioncolumns(dbmeta, 'cata')  
v1 = versioncolumns(dbmeta, 'cata', 'sch')  
v1 = versioncolumns(dbmeta, 'cata', 'sch', 'tab')
```

## Description

`v1 = versioncolumns(dbmeta, 'cata')` returns the list of all columns that are automatically updated when any row value is updated, for the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`v1 = versioncolumns(dbmeta, 'cata', 'sch')` returns the list of all columns that are automatically updated when any row value is updated, for the schema `sch`, in the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

`v1 = versioncolumns(dbmeta, 'cata', 'sch', 'tab')` returns the list of all columns that are automatically updated when any row value is updated, in the table `tab`, for the schema `sch`, in the catalog `cata`, for the database whose database metadata object is `dbmeta`, where `dbmeta` was created using `dmd`.

## Examples

Type

```
v1 = versioncolumns(dbmeta, 'orcl', 'SCOTT', 'BONUS', 'SAL')
```

MATLAB returns

```
v1 =  
    {}
```

In this example:

- `dbmeta` is the database metadata object.
- `orcl` is the catalog `cata`.
- `SCOTT` is the schema `sch`.
- `BONUS` is the table `tab`.
- `SAL` is the column name `l`.

The results show an empty set, meaning no columns automatically update when any row value is updates.

# versioncolumns

---

## See Also

columns, dmd, get



**Purpose** Get field size of column in fetched data set

**Syntax** `colsize = width(curs, colnum)`

**Description** `colsize = width(cursor, colnum)` returns the field size of the specified column number `colnum`, in the fetched data set `curs`.

**Examples** Get the width of the first column of the fetched data set, `curs`:

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

```
colsize =
```

```
11
```

The field size of column one is 11 characters (bytes).

**See Also** `attr`, `cols`, `columnnames`, `fetch`, `get`

# width

---

**Symbols**

[ ] 3-40  
 { } 3-40, 3-41

**A**

Advanced query options in VQB 2-27  
 All option in VQB 2-27  
 annotation  
   chart 2-22  
 annotation  
   display 2-19  
 Apply in VQB 2-31  
 array  
   data format 4-92  
 array format  
   Database Toolbox 2-12  
 attr 3-10, 4-10  
 Attributes 4-45  
 attributes of data 3-10, 4-10  
 AutoCommit 3-14, 4-44, 4-87

**B**

bestrowid 4-12  
 braces, curly 3-40, 3-41  
 brackets, square 3-40  
 bridge, JDBC/ODBC 1-4

**C**

Catalog 4-44  
 catalog, changing 4-35  
 CatalogName 4-47  
 cell array  
   data format 4-92  
 cell array data format 2-11

cell arrays  
   assigning values to cells 3-13  
   Database Toolbox 3-36  
   for exporting data 3-14  
   for query results 3-5  
 celldisp 3-40  
 Charting dialog box 2-20  
   data (x, y, z, and color) 2-21  
   Display 2-22  
   legend 2-21  
   preview 2-21  
   types of charts 2-20  
 charting query results 2-20  
 classpath.txt file 1-11  
 clearing variables from Data area 2-15  
 clearwarnings 4-13  
 close 3-10, 3-18, 4-14  
 cols 3-9, 4-16  
 ColumnCount 4-47  
 ColumnName 4-47  
 columnnames 3-9, 3-20, 4-17  
 columnprivileges 4-18  
 columns  
   attributes 3-10  
   automatically updated 4-109  
   cross reference 4-24  
   exported keys 4-36  
   foreign key information 4-51  
   imported key information 4-51  
   names 3-9, 3-14, 4-10, 4-17, 4-20  
   number 4-16  
   optimal set to identify row 4-12  
   primary key information 4-73  
   privileges 4-18  
   width 3-9, 4-111  
 columns 4-20

- ColumnTypeName 4-47
- columnWidth 4-10
- commit 3-14, 4-22
  - via exec 4-35
- Compiler, MATLAB ix
- Condition in VQB 2-31
- confds 1-12, 4-23
- Configure Data Source dialog box 4-23
- connection
  - clearing warnings for 4-13
  - closing 3-18, 4-14
  - creating 4-27
  - database, opening (establishing) 3-3, 4-27
  - information 4-71
  - JDBC 4-44
  - messages 4-44
  - object 3-3
  - opening 4-27
  - properties 4-43, 4-86
  - read-only 4-65
  - status 3-4, 4-71
  - time allowed for 3-2, 4-67
  - validity 4-60
  - warnings 4-44
- constructor functions 3-33
- converting numeric array to cell array 3-42
- crossreference 4-24
- currency 4-10
- Current clauses area in VQB 2-31
- Cursor 4-45

- cursor
  - attributes 4-45
  - closing 3-18, 4-14
  - creating via exec 4-33
  - creating via fetch 4-39
  - data element 4-45
  - error messages 4-45
  - importing data 3-5
  - object 3-4, 4-39
  - opening 3-4
  - properties 4-43, 4-86
  - resultset object 4-82

## D

- Data 4-45
- data
  - attributes 3-10, 4-10
  - cell array 3-14
  - column names 3-9, 4-17
  - column numbers 3-9, 4-16
  - committing 4-22, 4-87
  - displaying results in VQB 2-16
  - exporting 3-15, 4-56
  - field names 4-17
  - importing 3-5, 4-39
  - information about 3-8
  - inserting into database 3-22
  - replacing 3-17, 3-18, 4-107
  - retrieving from cell array 3-39
  - rolling back 4-83, 4-87
  - rows 3-8, 4-84
  - types vii, 1-5
  - updating 4-107
- Data area in VQB 2-8, 2-15

- data format 4-92
  - Database Toolbox 2-12
  - for query results 2-15, 4-39
  - preferences for retrieval 4-92
- data format, preferences 2-11
- data source
  - definition 1-7
  - for connection 4-27
  - ODBC connection 4-44
  - selecting for VQB 2-7
  - setting up 1-7
    - JDBC 1-11, 4-23
    - local ODBC 1-7
- data type 4-10
- database
  - connecting to 3-3, 4-27
  - JDBC connection 4-44
  - metadata object
    - creating 4-30
    - functions 3-29
    - properties 4-43
    - properties supported 4-102
  - name 4-27
  - supported 1-3
  - URL 4-27
- database 3-3
- Database Toolbox
  - about vi
  - features vii
  - installing 1-6
  - relationship of functions to VQB 2-5
  - starting 1-13
- DatabaseObject 4-45
- dbdemos 3-1
- demos 3-1
  - dbinfodemo 3-8
  - dbinsertdemo 3-12
  - dbupdatedemo 3-17
  - Visual Query Builder 2-6
- displaying
  - chart 2-22
  - query results
    - as chart 2-20
    - as report 2-23
    - in Report Generator 2-25
    - relationally 2-16
- Distinct option in VQB 2-27
- dmd 3-23, 4-30
- documentation
  - HTML xi
  - PDF xi
- dotted line in display of results 2-18
- driver 3-30, 4-31, 4-44
- driver object
  - functions 3-30, 3-32, 4-6
  - properties 3-31
- drivermanager 3-31, 4-32
- drivermanager object 3-30, 3-31
  - properties 4-43, 4-86
- Drivers 4-46
- drivers
  - JDBC 1-4
  - JDBC compliance 4-62
  - loading 4-81
  - ODBC 1-4
  - properties 4-32, 4-43
  - supported 1-4
  - unloading 4-106
  - validity 4-61
  - versions 3-31

**E**

- editing clauses in VQB 2-32
- error handling, preferences 2-11
- error messages 4-44, 4-45
- error notification, preferences 4-92
- examples
  - using functions 3-1
  - using VQB 2-5
- exec 3-4, 3-20, 4-33
- executing queries 2-8, 3-4, 3-20, 4-33
- exportedkeys 4-36
- exporting data
  - cell arrays 3-13
  - inserting 3-11, 3-15, 3-22, 4-56
  - replacing 3-17, 3-18, 4-107

**F**

- feature 1-13
- Fetch 4-45
- fetch 3-5, 3-37, 4-39
- fieldName 4-10
- fields
  - names 4-20
  - selecting for VQB 2-8
  - size (width) 3-9, 4-10, 4-111
- foreign key information 4-24, 4-36, 4-51
- format for data retrieved, preferences 4-92
- freeing up resources 4-14
- functions
  - database metadata object 3-29
  - driver object 3-32

**G**

- get 3-14, 3-31, 4-43
- grouping statements 2-34
  - removing 2-38

**H**

- Handle 4-44
- help
  - online xi
  - Visual Query Builder 2-6
- HTML documentation xi
- HTML report of query results 2-23, 2-25

**I**

- importedkeys 4-51
- importing data
  - using functions 3-2, 3-4, 3-5, 4-39
  - using VQB 2-7
- index for resultset column 4-70
- indexinfo 4-54
- insert 3-15, 4-56
- inserting data into database 3-22
- installing Database Toolbox 1-6
- Instance 4-44
- isconnection 4-60
- isdriver 3-31, 4-61
- isjdbc 4-62
- isNullable 4-47
- isnullcolumn 4-63
- isReadOnly 4-47
- isreadonly 4-65
- isurl 4-66

- J**
- Java Database Connectivity. *See* JDBC
  - JDBC
    - compliance 4-62
    - connection object 4-44
    - driver instance 4-44
    - drivers
      - names 4-27
      - supported 1-4
      - validity 4-61
    - setting up data source 1-11
    - SQL conversion to native grammar 4-101
    - URL 4-27, 4-44
  - JDBC/ODBC bridge 1-4
  - join operation in VQB 2-47
- L**
- legend
    - in chart 2-21
    - labels in chart 2-21
  - loading saved queries 2-11
  - LoginTimeout 3-32, 4-44, 4-46
  - logintimeout 3-2, 4-67
  - LogStream 4-46
- M**
- MajorVersion 4-45
  - MATLAB
    - version 1-2
    - workspace variables in VQB 2-8
  - MATLAB Compiler ix
  - memory problems
    - RowLimit solution 4-39
  - Message 4-10, 4-44, 4-45
  - metadata object
    - database 3-23, 4-30
    - database functions 3-29
    - resultset 4-85
    - resultset functions 3-29
  - methods 3-33
  - M-files 3-1
  - MinorVersion 4-45
  - multiple entries, selecting 2-8
- N**
- namecolumn 4-70
  - NULL values
    - detecting in imported record 4-63
    - function for handling 2-14
    - preferences for reading and writing 2-11
    - reading from database 3-19
    - representation in results 2-12
    - writing to database 2-11
  - null values
    - preferences for reading and writing 4-92
  - nullable 4-10
  - num2cell 3-42
  - numeric data format 2-11, 4-92
- O**
- objects 3-33
    - creating 3-33
    - properties, getting 4-43
  - ObjectType 4-44
  - ODBC
    - setting up data source 1-7
  - ODBC drivers
  - online help xi, 2-6
  - Open Database Connectivity. *See* ODBC drivers

Operator in VQB 2-32  
ORDER BY Clauses dialog box 2-39  
Order by option in VQB 2-38  
overloaded functions 3-34

## P

parentheses, adding to statements 2-34  
password 3-3, 4-27  
PDF documentation xi  
ping 3-4, 3-14, 4-71  
platforms 1-2  
precision 4-10  
preferences  
    for Visual Query Builder 2-11  
primary key information 4-24  
primarykeys 4-73  
printing  
    chart 2-22  
    display 2-19  
    report 2-24  
privileges  
    columns 4-18  
    tables 4-104  
procedurecolumns 4-75  
procedures 4-77  
properties  
    database metadata object 3-24, 4-102  
    driver 3-31  
    getting 4-43  
    setting 4-86

## Q

qry file extension 2-10  
queries  
    accessing values in multiple tables 2-42, 2-47  
    creating with VQB 2-7  
    displaying results  
        as chart 2-20  
        as report 2-23  
        in Report Generator 2-25  
        relationally 2-16  
    executing 2-8  
    loading saved queries 2-11  
    ordering results 2-38  
    refining 2-29  
    results 2-8, 3-34, 4-45  
    running via exec 4-33  
    saving 2-10  
    select statement 3-4  
    viewing results 2-9  
querybuilder 4-79  
querytimeout 4-80  
quitting  
    Visual Query Builder 2-4

## R

ReadOnly 4-44  
readOnly 4-10  
refining queries 2-29  
register 4-81  
Relation in VQB 2-31  
relational display of query results 2-16  
replacing data 3-17, 3-18, 4-107  
Report Generator display of query results 2-25  
reporting query results 2-23, 2-25  
requirements, system 1-2  
reserved words 1-3



- results
    - from query 2-8
    - viewing 2-9
  - ResultSet 4-45
  - resultset
    - clearing warnings for 4-13
    - closing 4-14
    - column name and index 4-70
    - metadata object 3-29
      - creating 4-85
      - properties 4-43
    - object, functions 4-6
    - properties 4-43
  - resultset 4-82
  - retrieving
    - data from cell arrays 3-39
    - data from database 2-7
  - rollback 4-83
  - RowLimit 4-39, 4-45, 4-88
  - rows 3-8, 4-84
  - rows, uniquely identifying 4-12
  - rsmd 4-85
  - running queries 2-8
- S**
- saving queries 2-10
  - scale 4-10
  - select statement 3-4
  - selecting data from database 4-33
  - selecting multiple entries in VQB 2-8
  - set 3-32, 4-86
  - setdbprefs 2-14, 3-19, 4-92
  - size 3-20
  - size of field 3-9
  - Sort key number in VQB 2-39
  - Sort order in VQB 2-39
  - spaces in table and column names 1-3
  - SQL
    - commands 1-4
    - conversion to native grammar 4-101
    - join in VQB 2-47
    - statement
      - executing 4-33
      - in exec 3-4, 3-18, 4-45
      - in VQB 2-33
      - time allowed for query 4-80
      - where clause 3-18, 4-107
  - sql2native 4-101
  - SQLQuery 4-45
  - starting
    - Database Toolbox 1-13
    - Visual Query Builder 1-13, 2-2, 2-7
  - Statement 4-45
  - status of connection 3-4, 4-71
  - stored procedures
    - in catalog or schema 4-77
    - information 4-75
    - running 4-35
  - string and numeric 4-92
  - structure data format 2-11, 4-92
  - subqueries
    - in VQB 2-42
  - Subquery dialog box 2-42
  - supports 3-26, 4-102
  - system requirements 1-2

**T**

- TableName 4-47
- tableprivileges 4-104
- tables
  - index information 4-54
  - names 4-105
  - privileges 4-104
  - selecting for VQB 2-8
  - selecting multiple for VQB 2-48
- tables 3-28, 4-105
- time
  - allowed for connection 4-67
  - allowed for SQL query 4-80
- TimeOut 4-44
- TransactionIsolation 4-44
- tutorial
  - Visual Query Builder 2-5
- Type 4-45
- typeName 4-10
- typeValue 4-10
- typographical conventions (table) xii

**U**

- undo 3-14
- ungrouping statements 2-38
- unique occurrences of data 2-27
- unregister 4-106
- update 3-18, 4-107
- URL
  - JDBC database connection 4-27
  - validity 4-66
- URL 4-44
- UserName 4-44
- username 3-3, 4-27

**V**

- versioncolumns 4-109
- viewing query results 3-37
- Visual Query Builder
  - demo 2-6
  - examples 2-5
  - help 2-6
  - overview 2-2
  - quitting 2-4
  - relationship to Database Toolbox functions 2-4
  - starting 1-13, 2-2, 2-7, 4-79
  - steps to use 2-3
- VQB. *See* Visual Query Builder

**W**

- Warnings 4-44
- warnings, clearing 4-13
- where clause 3-18, 4-107
- WHERE Clauses dialog box 2-30
- Where option in VQB 2-29
- width 3-9, 4-111
- workspace variables in VQB 2-8
  - clearing from Data area 2-15
- writable 4-44