Database Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

User's Guide

Version 2.1

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

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

`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`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
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

Database Toolbox User's Guide

Printing History:

May 1998	Version 1 release for MATLAB 5.2 (online only)
July 1998	First printing for Version 1 (Releases 10 and 11)
June 1999	Updated for Version 2 for Release 11 (online only)
December 1999	Second printing (Release 11, Version 2)
September 2000	Updated for Version 2.1 for Release 12 (online only)

Preface

Installation and Setup

1

2 Visual Query Builder Tutorial

Tutorial for Functions

3

Function Reference

4

Preface

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 MATLAB's 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 as a string, parsed into the correct data types, and stored in a MATLAB cell array. At that point, you use MATLAB's extensive set of 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 Database Toolbox also comes with the Visual Query Builder (VQB), 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 a MATLAB cell array so you then can process the data using MATLAB's suite of functions. With the VQB, you can display the retrieved information in relational tables, reports, and charts.

New Features

Version 2.1

- The Database Toolbox now runs on all platforms that support MATLAB 6, with the exception of the HP 10.2 (HP 700).

- Performance for fetching data from your database has increased by a factor of roughly 100 over Version 2.0. This improvement was first introduced in Version 2.0.1.
- Do not run `feature('dispatchjava',1)` to start the Database Toolbox, as was required for Version 2.0. Instead, begin by running the Database Toolbox function you want to use.
- When using the Visual Query Builder, you can now export query results using the Report Generator, if the Report Generator product is installed locally. To use it, select **Report Generator** from the Visual Query Builder **Display** menu.
- A **Group** button has been added to the **Where**, **Subquery**, and **Having** dialog boxes. Use the **Group** button to group constraints for a single field, especially when using the `OR` operator. Basically, the **Group** button allows you to evaluate a set of constraints as a whole.

Version 2

Version 2 of the Database Toolbox includes these new features:

- The Visual Query Builder, an easy-to-use graphical user interface for retrieving data from your database.
- Support for UNIX – You can now run the Database Toolbox on UNIX platforms. Note that for MATLAB Release 12, the Database Toolbox does not run on the HP 10.2 platform.
- Over 30 new functions – These include an invaluable set of functions for retrieving database metadata so you can find out information about the database, for example, table names and column names. Other new functions are for drivers and resultsets.
- To use the new version of the database toolbox, you need to run the command `feature('dispatchjava',1)`. (Note that this is not required for the Database Toolbox on MATLAB Release 12.)

Features of the Database Toolbox

The Database Toolbox has the following features:

- Data types are automatically preserved in MATLAB – No data massaging or manipulation is required. The data is stored in MATLAB cell arrays, which support mixed data types.
- 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.
- 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.
- 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.
- 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.
- 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 functions.
- 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 following toolboxes all include functions that extend MATLAB's capabilities.

Product	Description
Data Acquisition Toolbox	MATLAB functions for direct access to live, measured data from MATLAB
Datafeed Toolbox	MATLAB functions for integrating the numerical, computational, and graphical capabilities of MATLAB with financial data providers
Financial Time Series Toolbox	Tool for analyzing time series data in the financial markets
Financial Toolbox	MATLAB functions for quantitative financial modeling and analytic prototyping

Product	Description (Continued)
GARCH Toolbox	MATLAB functions for univariate Generalized Autoregressive Conditional Heteroskedasticity (GARCH) volatility modeling
MATLAB Runtime Server	MATLAB environment in which you can take an existing MATLAB application and turn it into a stand-alone product that is easy and cost-effective to package and distribute. Users access only the features that you provide via your application's graphical user interface (GUI) - they do not have access to your code or the MATLAB command line.

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 Chapter 2, "Visual Query Builder Tutorial."

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-based 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 Chapter 3, "Tutorial for Functions", and Chapter 4, "Function Reference."

Organization of the Document

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

Section	Description
Chapter 1, "Installation and Setup"	Provides system requirements and describes how to install the Database Toolbox and set up an ODBC data source or a JDBC driver.
Chapter 2, "Visual Query Builder Tutorial"	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 97. If you have this version of Microsoft Access installed on your system, you can perform the steps exactly as shown.

Section	Description (Continued)
Chapter 3, "Tutorial for Functions"	Presents examples with instructions for using many of the Database Toolbox functions. The tutorial uses a sample database, `Northwind`, that is distributed with Microsoft Access. 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 book uses the following typographical conventions.

Item	Convention to Use	Example
Example code	`Monospace font`	To assign the value 5 to `A`, enter `A = 5`

MATLAB output	`Monospace font`	MATLAB responds with
		`A =`
		`  5`

Function names and syntax	`Monospace font`	The `close` function uses the syntax: `close(cursor)`
Literal strings (in syntax) must be typed as is	`Monospace bold` for literals.	`set(conn, 'AutoCommit', 'value')`
String variables having a prescribed set of values	*`Monospace italics`*	`set(conn, 'AutoCommit', 'value')` where `'value'` can be `on` or `off`
Mathematical expressions	Variables in *italics* Functions, operators, and constants in standard text.	This vector represents the polynomial $p = x^2 + 2x + 3$
Menu names, menu items, and controls	**Boldface** with an initial capital letter	Choose the **File** menu.
Keys	**Boldface** with an initial capital letter	Press the **Enter** key.
New terms	*Italics*	An *array* is an ordered collection of information.

In addition, some words in our syntax lines are shown within single quotation marks. The single quotation marks are a MATLAB requirement and must be typed. For example

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

1

Installation and Setup

System Requirements

The Database Toolbox 2.1 works with the following systems and applications:

Platforms

The Database Toolbox 2.1 runs on all of the platforms that support MATLAB Release 12 and Java. The Database Toolbox 2.1 does not run on the Hewlett-Packard 10.2 platform.

MATLAB Version

The Database Toolbox 2.1 requires MATLAB Version 6 (Release 12) 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 for the following database management systems:

- IBM DB2 Universal Version 5
- Informix Version 7.2.2
- Ingres
- Microsoft Access 95 or 97
- Microsoft SQL Server Version 6.5 or 7.0
- Oracle Version 7.3.3
- Sybase SQL Server Version 11.0
- Sybase SQL Anywhere Version 5.0

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 verison.

Drivers

For PC platforms, the Database Toolbox supports Open Database Connectivity (ODBC) drivers used with the supported databases,. For UNIX and PC platforms, the Database Toolbox supports Java Database Connectivity (JDBC) drivers.

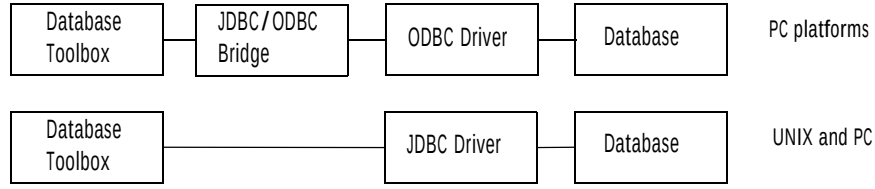
The driver for your database must be installed in order to use the Database Toolbox. Most users (or their database administrators) install the driver when they install the database. Consult your database documentation if you need instructions to install a database driver.

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
- 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* types of MATLAB data not on this list to a database, 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.

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 on 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 slightly depending on your configuration. Use one of these sets of instructions:

- For MATLAB PC platforms whose database resides on that PC, use "Setting Up a Local Data Source for ODBC Drivers" on page 1-6.
- For MATLAB PC platforms whose database resides on another system to which the PC is networked, use "Setting Up a Remote Data Source for ODBC Drivers" on page 1-8.
- For MATLAB platforms that connect to a database via a JDBC driver, use "Setting Up a Data Source for JDBC Drivers" on page 1-12.

Setting Up a Local Data Source for ODBC Drivers

Follow this procedure to set up a local data source for a PC. This procedure uses as an example, the Microsoft ODBC driver Version 4.00.42 and the U.S. English version of Microsoft Access 97 for Windows NT. 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`
- `SampleDB` data source – Uses the Microsoft Access sample database called `Northwind`

1. From the Windows **Start** menu, select **Control Panel** from the **Settings** menu.
2. Double-click **ODBC Data Sources**.

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

3 Select the **User DSN** tab.

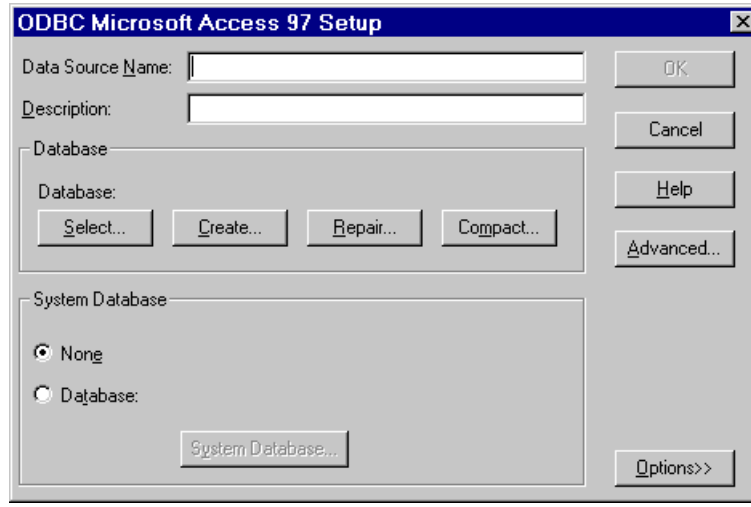
A list of existing system data sources appears.

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

5 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**.
- 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.



6 Provide a **Data Source Name** and **Description**.

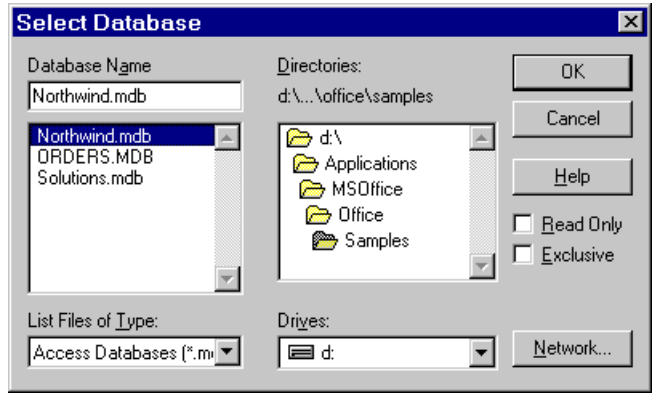
For the first example data source, type `dbtoolboxdemo` as the data source name. For the other example data source, type `SampleDB` as the data source name.

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

7 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`.

      For the `SampleDB` data source, select `Northwind.mdb` in the `msoffice\...\Samples` directory.

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

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

9 Click **OK** to close the **ODBC Data Source Administrator** dialog box.

Setting Up a Remote Data Source for ODBC Drivers

Follow this procedure to set up a data source that resides on a remote system to which your PC has network access. This procedure uses the Microsoft ODBC driver Version 4.00.42 and the U.S. English version of Microsoft Access 97 for Windows NT installed on a networked server. If you have a different configuration, you may have to modify the instructions.

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

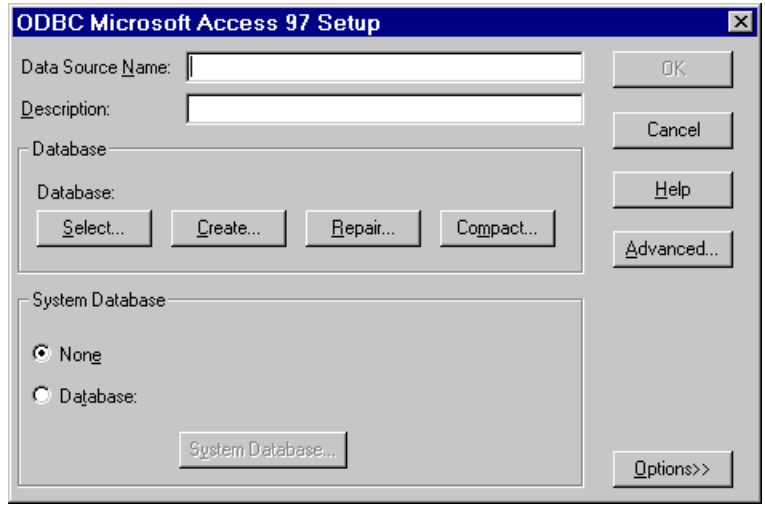
- `dbtoolboxdemo` data source – Uses the Microsoft Access `tutorial` database provided with the Database Toolbox in `$matlabroot\toolbox\database\dbdemos`
- `SampleDB` data source – Uses the Microsoft Access sample database called `Northwind`

1. From the Windows **Start** menu, select **Control Panel** from the **Settings** menu.
2. Double-click **ODBC**.

   The **ODBC Data Source Administrator** dialog box appears.
3. Select the **System DSN** tab.

   A list of existing system data sources appears.
4. Click **Add**. A list of installed ODBC drivers appears in the **Create New Data Source** dialog box.
5. Select the ODBC driver that the remote data source you are creating will use and click **Finish**.
   - For the examples in this book, select **Microsoft Access Driver**.
   - 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 one shown in the following figure.



6 Provide a **Data Source Name** and **Description**.

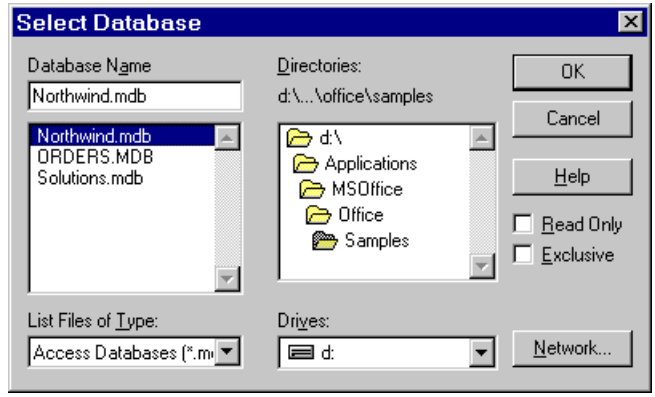
For one of the example data sources, type `dbtoolboxdemo` as the data source name. For the other example data source, type `SampleDB` as the data source name.

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

7 Select the database that this data source will use. Note that for some drivers, you skip this step.

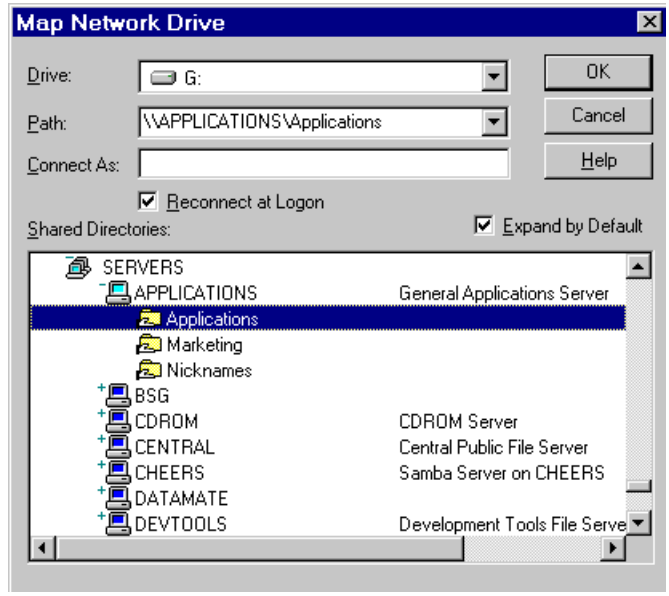
a In the **ODBC Setup** dialog box, click **Select**.

The **Select Database** dialog box appears.



b Click **Network**.

The **Map Network Drive** dialog box appears.



c Find and select the directory containing the database you want to use, and then click **OK**. The **Map Network Drive** dialog box closes.

For the `dbtoolboxdemo` data source, select the `$matlabroot\toolbox\database\dbdemos` directory.

For the `SampleDB` data source, select the `msoffice\...\Samples` directory.

In the example shown, the database is in `SERVERS\APPLICATIONS\Applications`.

d Locate the database in the **Select Database** dialog box.

For the `dbtoolboxdemo` data source, select `tutorial.mdb`. For the `SampleDB` data source, select `Northwind.mdb`.

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

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

9 Click **OK** to close the **ODBC Data Source Administrator** dialog box.

Setting Up a Data Source for JDBC Drivers

1 To set up a data source for use with a UNIX workstation or PC using JDBC drivers, you 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
```

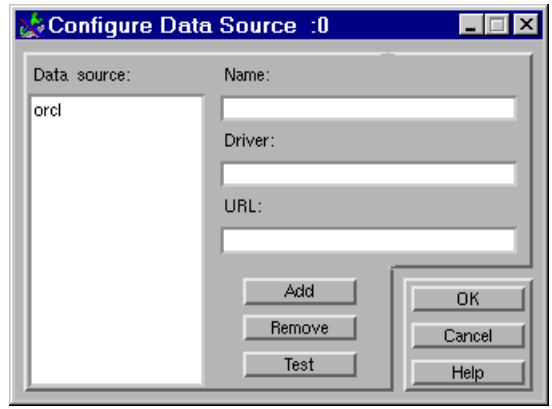
where `classes111.zip` is the file containing JDBC drivers. The file is available from your database provider.

2 If you want to use the Visual Query Builder, perform these steps after completing step 1 to set up the JDBC data source.

a Start MATLAB if it is not already running.

b Access the **Configure Data Source** dialog box by typing

```
confds
```



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

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

Name: `orcl`

Driver: `oracle.jdbc.driver.OracleDriver`

URL: `jdbc:oracle:thin:@144.212.33.130:1521:`

d Click **Add** to add the data source.

e 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.

f Click **OK** to save the changes and close the **Configure Data Source** dialog box.

To remove the 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

To use the Database Toolbox functions, just type the function you want to use. For more information, see “Tutorial for Functions” on page 3-1.

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

2

Visual Query Builder Tutorial

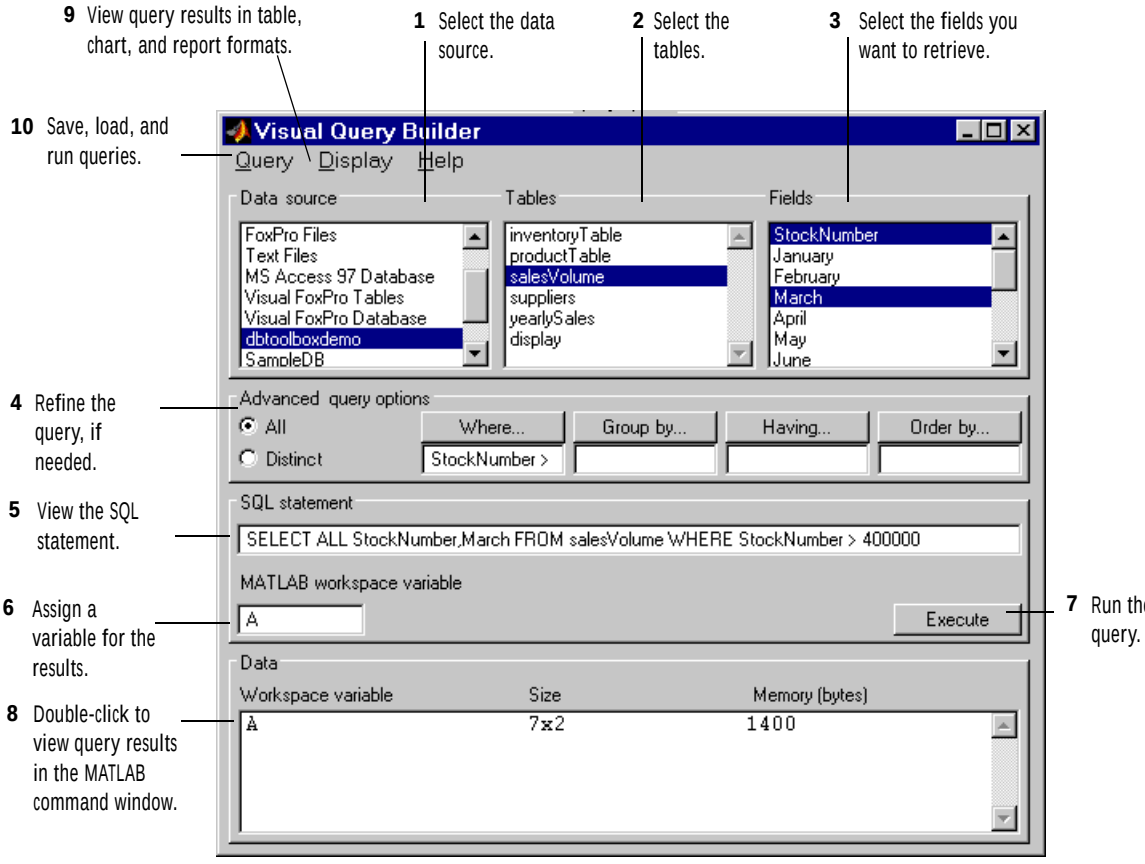
About the Visual Query Builder

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 so you can process the data using MATLAB's suite of functions. With the VQB, you can display the retrieved information in relational tables, reports, and charts.

Visual Query Builder Interface

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

The following illustration depicts the key features of the interface and points out the main steps that you perform, in order, from 1 through 10, to build and run a query using the interface. These steps are detailed in examples throughout this chapter.



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 build SQL queries easily and to import results into MATLAB, or use Database Toolbox functions instead.

When to Use Database Toolbox Functions

Use the Database Toolbox functions to:

- Export results from MATLAB to databases.
- Write MATLAB-based applications that access databases.
- Perform other functions not available with the Visual Query Builder.

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 Chapter 3, "Tutorial for Functions" and Chapter 4, "Function Reference."

Examples Using the Visual Query Builder

Many of the Visual Query Builder features are demonstrated in this book using simple examples. These examples use the `dbtoolboxdemo` data source (`tutorial` database). Instructions for setting up this data source are in Chapter 1, "Installation and Setup."

If your version of Microsoft Access is different than that referred to in "Installation and Setup", 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 tutorial.

The examples used are:

- "Starting and Quitting the Visual Query Builder" on page 2-6.
- "Building, Running, and Saving a Query" on page 2-7.
- "Viewing Query Results" on page 2-12.
- "Fine-Tuning Queries Using Advanced Query Options" on page 2-22.

Example in the Visual Query Builder Demo

In the **Visual Query Builder** dialog box, select **Demo** from the **Help** menu. This runs a demonstration of the main features of the VQB. The demo runs on PC platforms only. It uses the `dbtoolboxdemo` data source (`tutorial` database). Instructions for setting up this data source are in Chapter 1, "Installation and Setup".

If your version of Microsoft Access is different than that referred to in "Installation and Setup", 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 databases 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:

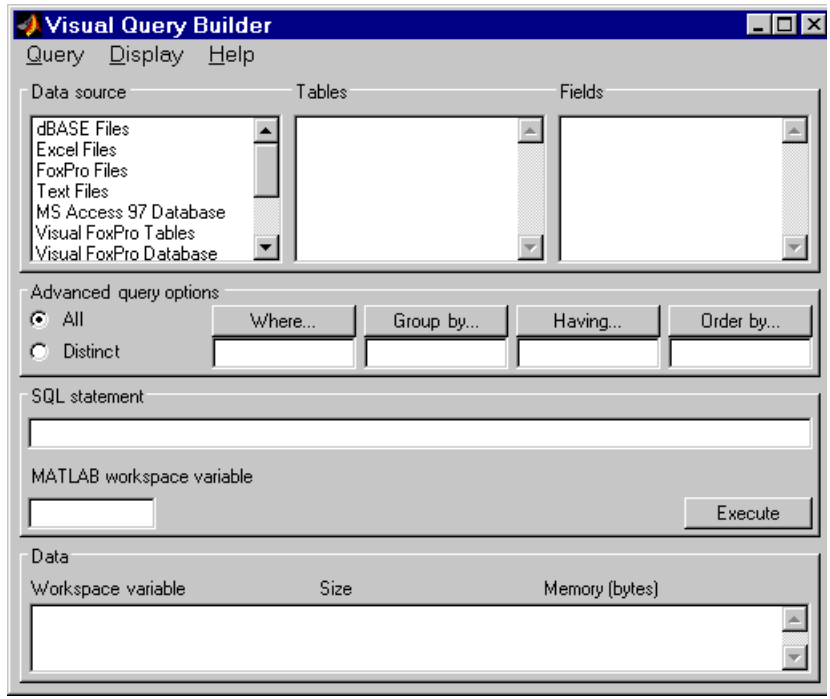
- Selecting **Visual Query Builder Help** from the **Help** menu. This tutorial for the Visual Query Builder appears in the Help browser.
- Clicking **Help** in any Visual Query Builder dialog box. Detailed instructions for that dialog box appear in the Help browser.

Starting and Quitting the Visual Query Builder

To start the Visual Query Builder, type

```
querybuilder
```

The **Visual Query Builder** dialog box appears.



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

Building, Running, and Saving a Query

Topics covered in this section are:

Building and Running a Query

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

1. Start the Visual Query Builder – see "Starting and Quitting the Visual Query Builder" on page 2-6.

2. 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" in Chapter 1.

   After selecting a data source, the list of tables in that data source appears.

3. 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.

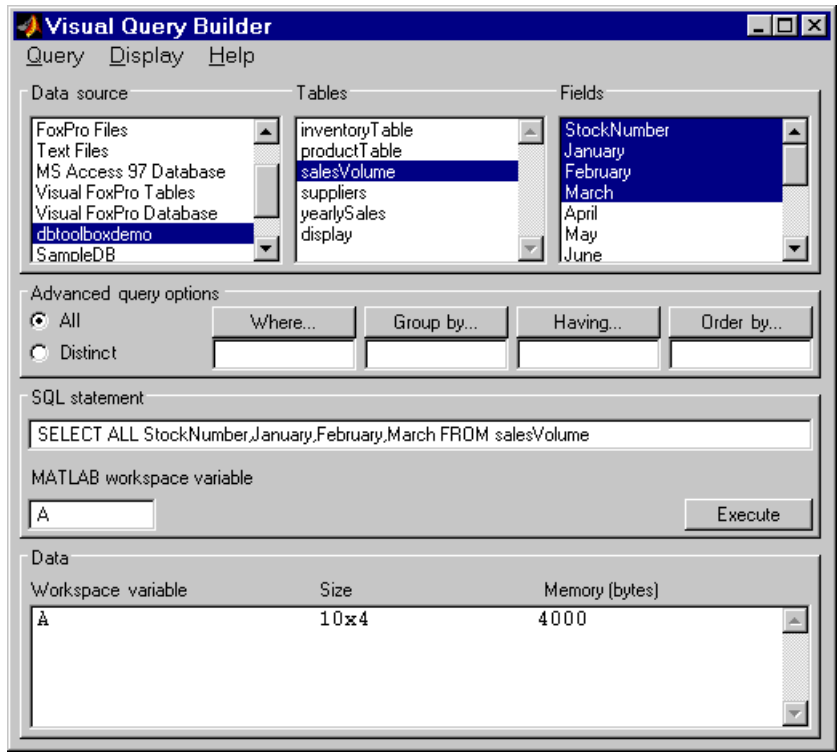
4. From the **Fields** list box, select the fields containing the data you want to import. To select more than one field, hold down the **Ctrl** key or **Shift** key while selecting. For this example, select the fields `StockNumber`, `January`, `February`, and `March`.

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

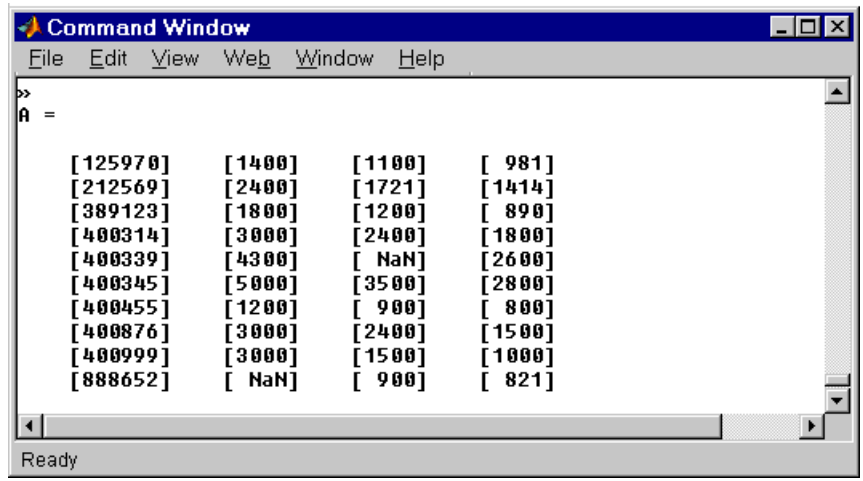
5 In the **MATLAB workspace variable** field, assign a name for the data returned by the query. For this example, use A.

6 Click **Execute** to run the query and retrieve the data.

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



7 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**.



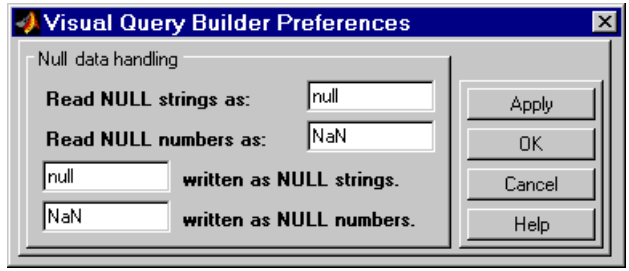
As an example of how to read the results, sales for item 400876 are 3000 in January, 2400 in February, and 1500 in March.

NULL Values

In the example results, there are two NaN values in the data, which represent NULLS. You can specify how you want the query builder to represent NULL data.

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

The **Preferences** dialog box appears, showing the current settings for handling `NULL` values.



2 Change values in the **Preferences** dialog box and click **OK**. For the example, change the value for **Read NULL numbers as:** from `NaN` to `0`.

3 Click **Execute** to run the query again.

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

The results show `0`'s where previously there were `NaN` values.

Preferences apply to the current MATLAB session.

Another way to set preferences is by using the `setdbprefs` command. To use the same preferences whenever you run MATLAB, include the `setdbprefs` command 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 as a cell array, where each element in the array points to an array that consists of a single value. The cell array structure allows a mixture of data types.

For more information about working with cell arrays, see "Working with Cell Arrays in MATLAB" on page 3-36.

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 Provide 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 Assign a **MATLAB workspace variable** and click **Execute** to run the query.

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 appear in the **Data** area when you execute a query. They remain in the **Data** area until you clear them in the **Command Window** using the `clear` command, and then execute a query.

Viewing Query Results

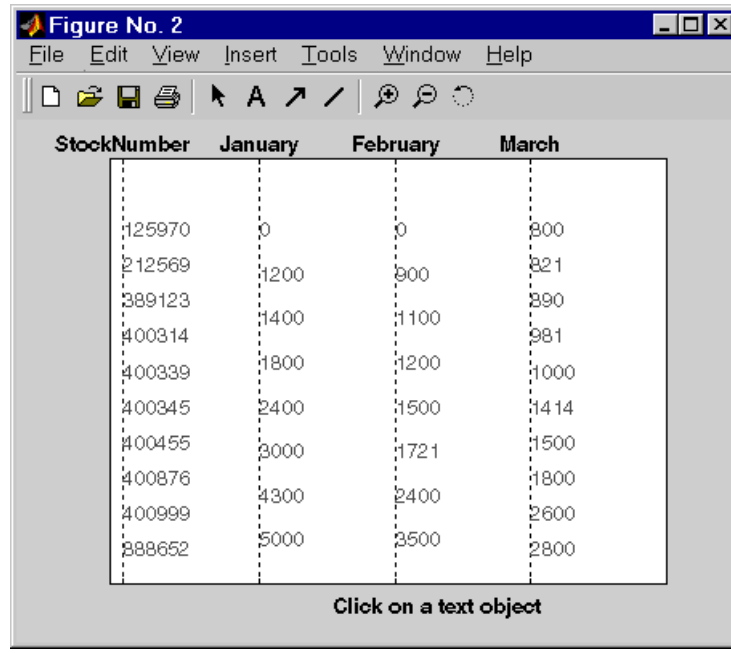
After running a query in the Visual Query Builder, you can view:

- The retrieved data in the MATLAB command window, as described in step 7 of "Building, Running, and Saving a Query" on page 2-7.
- A "Relational Display of Data" on page 2-13.
- A "Chart Display of Results" on page 2-16; for example, a pie chart.
- A "Report Display of Results in a Table" on page 2-19.
- A "Display of Results in the Report Generator" on page 2-20

Relational Display of Data

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

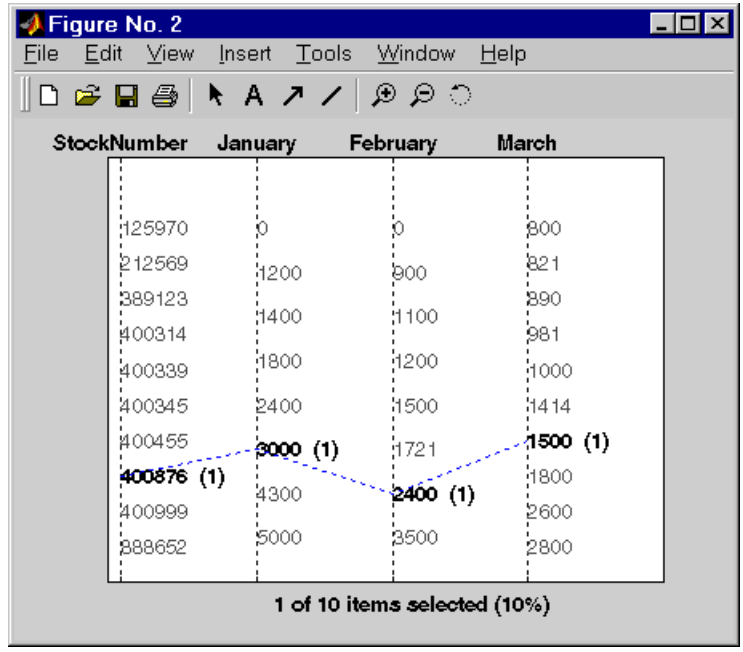
The query results appear in a figure window.



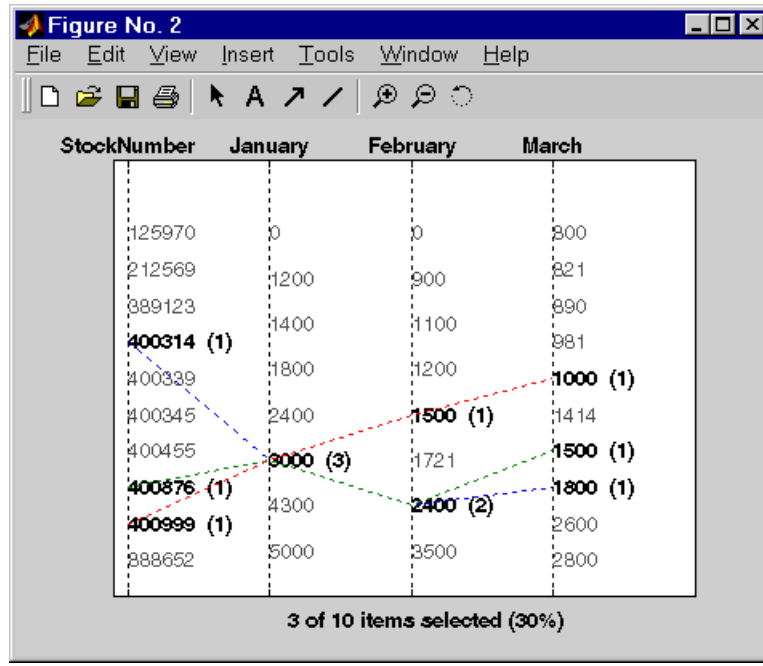
The display shows only the unique values for each field. For example, there are 10 entries in the `StockNumber` field, 8 entries in the `January` and `February` fields, and 10 entries in the `March` field, corresponding to the number of unique values in those fields. Therefore, do *not* read each row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

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. There are three different items with sales of 3000 units in January, 400314, 400876, and 400999.



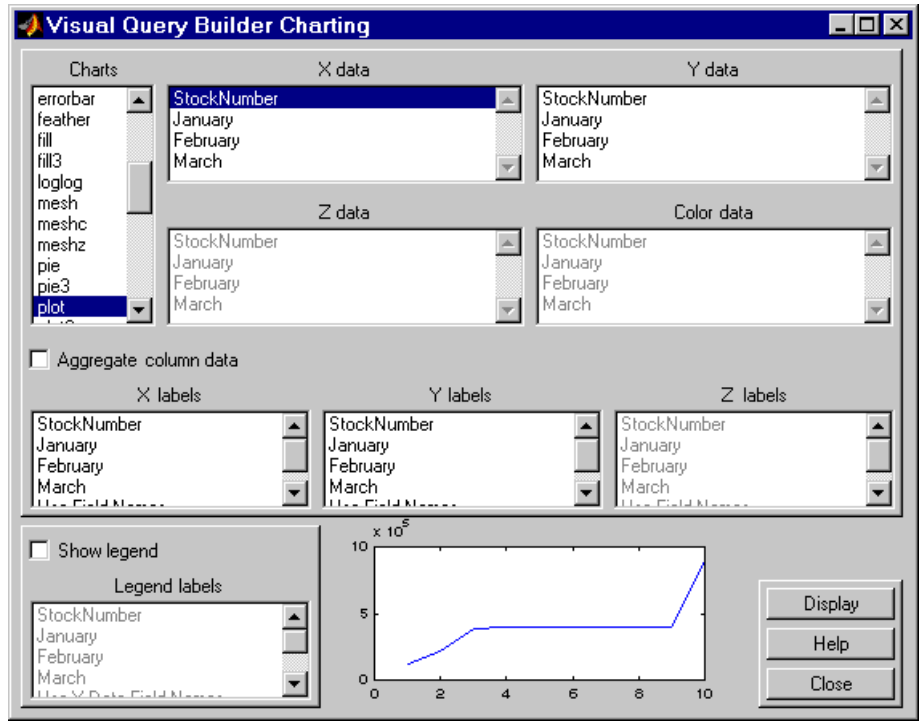
3 Because the display is presented in a MATLAB figure window, you can use some MATLAB figure functions. For example, you can print the figure and annotate it. For more information, use the **Figure** window's **Help** menu.

4 If the search results include many entries, the display might not effectively show all of them. You can stretch the window to make it larger, narrow the search so there are fewer results, or use "Report Display of Results in a Table" on page 2-19.

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. 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, it shows the pie chart, 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.

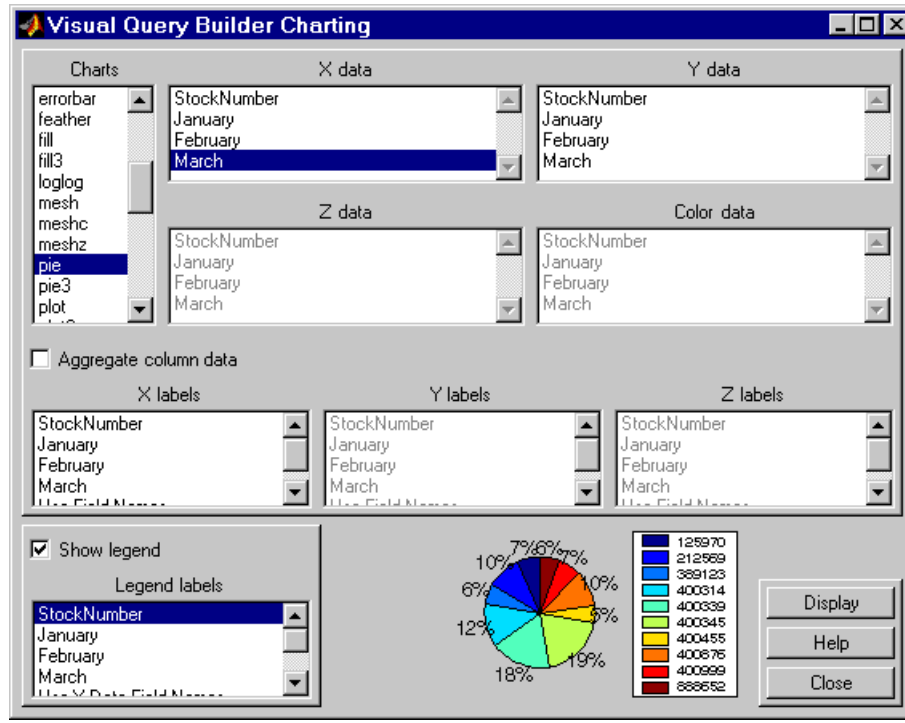
The preview of the chart at the bottom of the dialog box reflects the selection you made. For this example, the pie chart shows percentages for March data.

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

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.

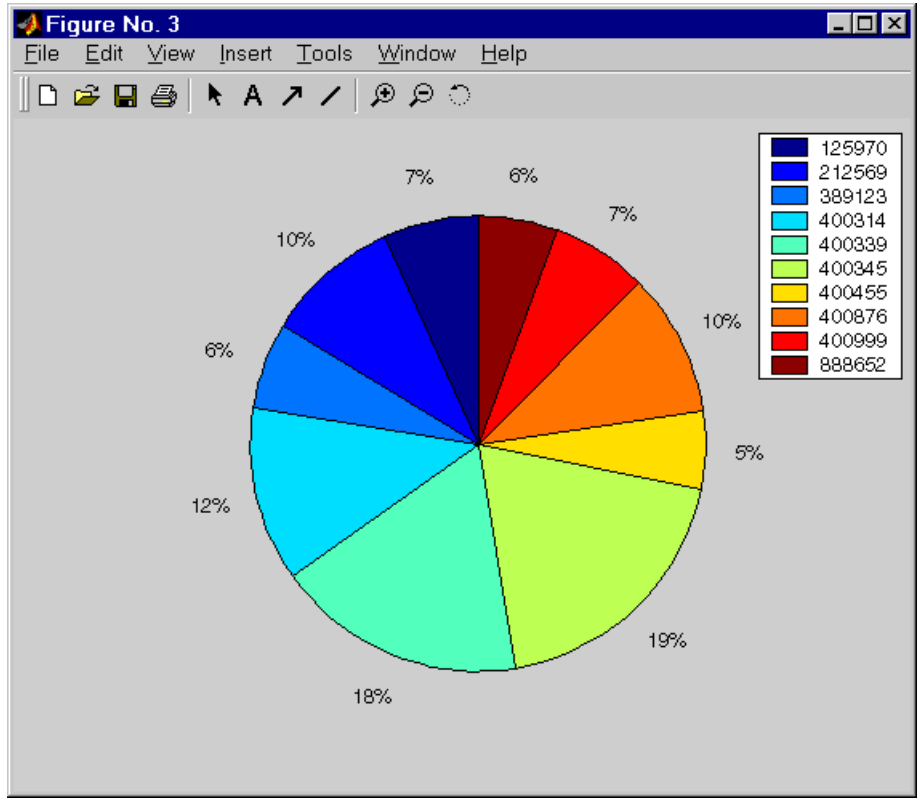


6 Click **Display**.

The pie chart appears in a figure window. Because the display is presented in a MATLAB figure window, you can use some MATLAB figure functions such as printing or annotating the figure. 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.

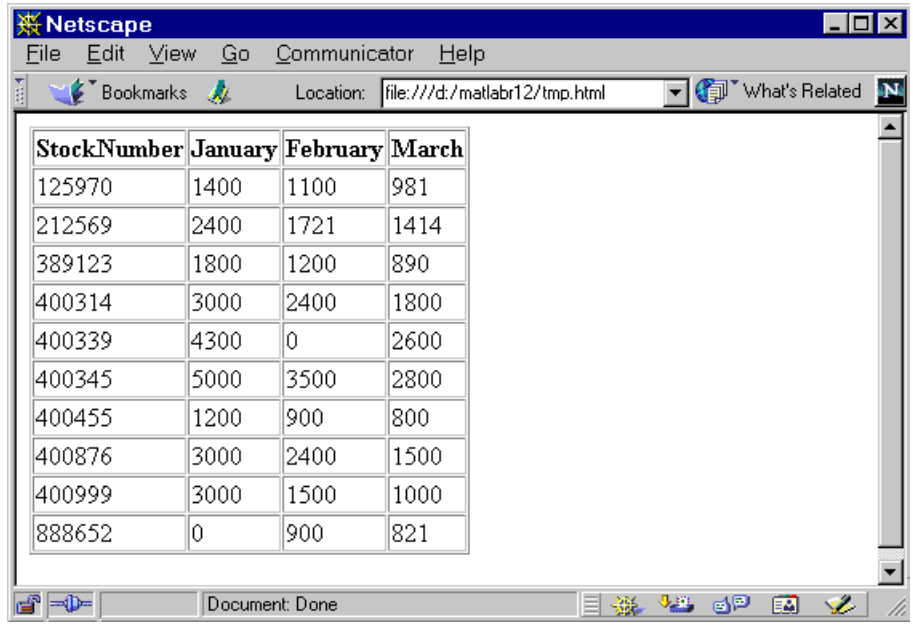
Report Display of Results in a Table

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 is slightly different.



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.

Display of Results in the Report Generator

You need to have the MATLAB Report Generator product installed locally on your system to use this option.

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.

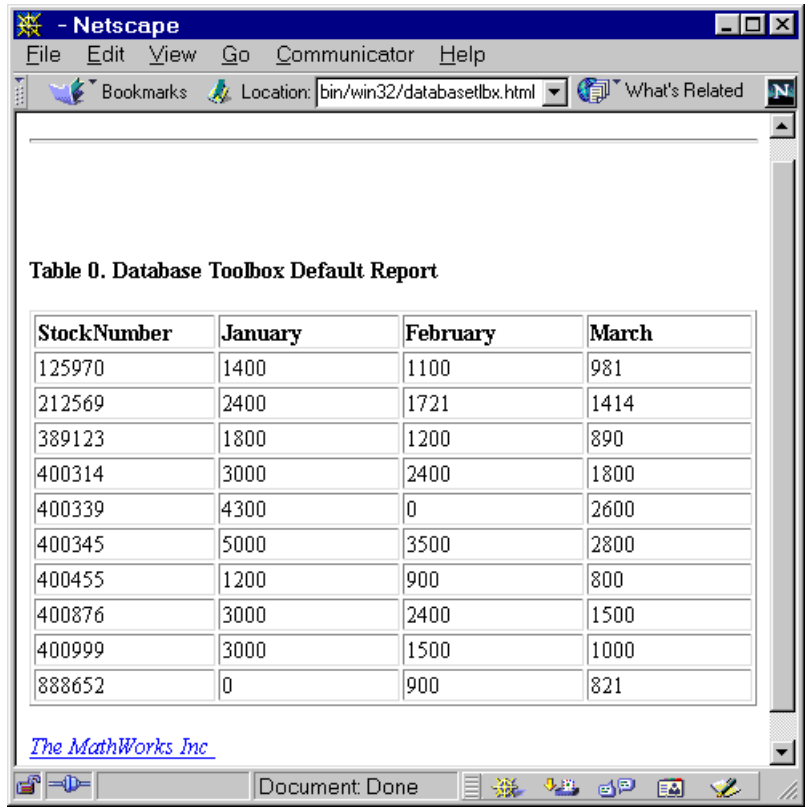
The **Setup File List** dialog box appears.

3 Select `databasetlbx.rpt` from the list.

4 To modify the report format, click **Edit**. See the help for the Report Generator for instructions.

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

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



This example shows a report of sales volume over three 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:

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

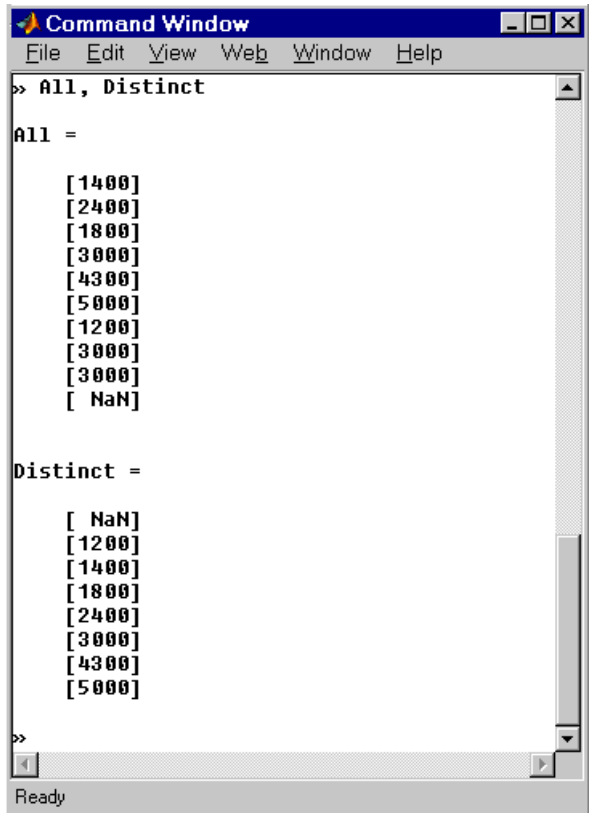
Retrieving Unique Occurrences

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. Select the **Data source**; for example, `dbtoolboxdemo`.
2. Select the **Tables**; for example, `SalesVolume`.
3. Select the **Fields**; for example, `January`.
4. Run the query to retrieve all occurrences.
   a. In **Advanced query options**, select **All**.
   b. Assign a **MATLAB workspace variable**; for example, `All`.
   c. Click **Execute**.
5. Run the query to retrieve only unique occurrences.
   a. In **Advanced query options**, select **Distinct**.
   b. Assign a **MATLAB workspace variable**, for example, `Distinct`.
   c. Click **Execute**.

6 In the **Data** area, the **Workspace variable size** shows `10x1` for `All` and `8x1` for `Distinct`.

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



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

Retrieving Information That Meets Specified Criteria

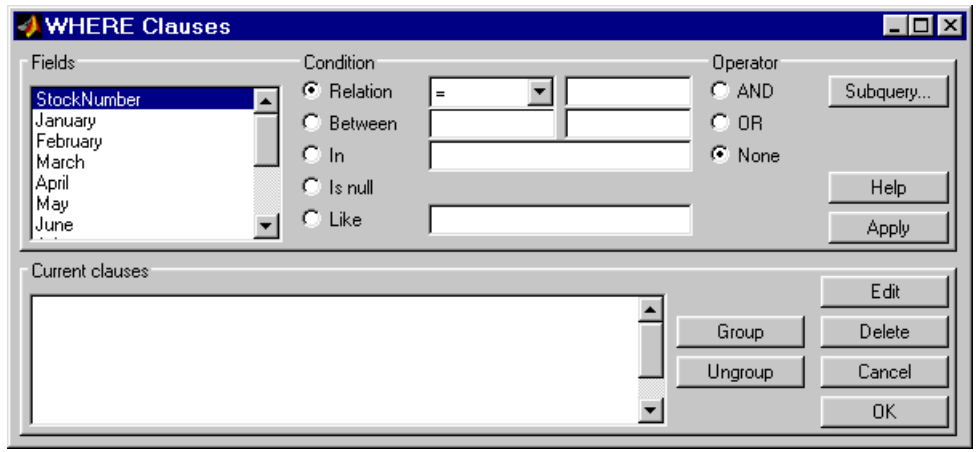
Use the **Where** field in **Advanced query options** to retrieve only the information that meets criteria you specify. This example uses `basic.qry`, created and saved in “Building, Running, and Saving a Query” on page 2-7. It

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

The **Where Clauses** dialog box appears.



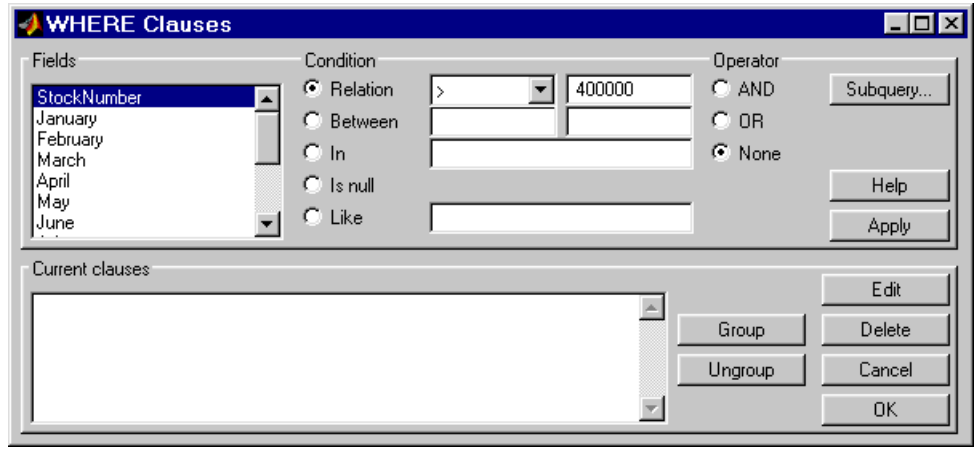
3 Select the **Fields** whose values you want to restrict. For example, select `StockNumber`.

4 Use **Condition** to specify the criteria. For example, specify that the `StockNumber` be greater than `400000`.

a Select **Relation**.

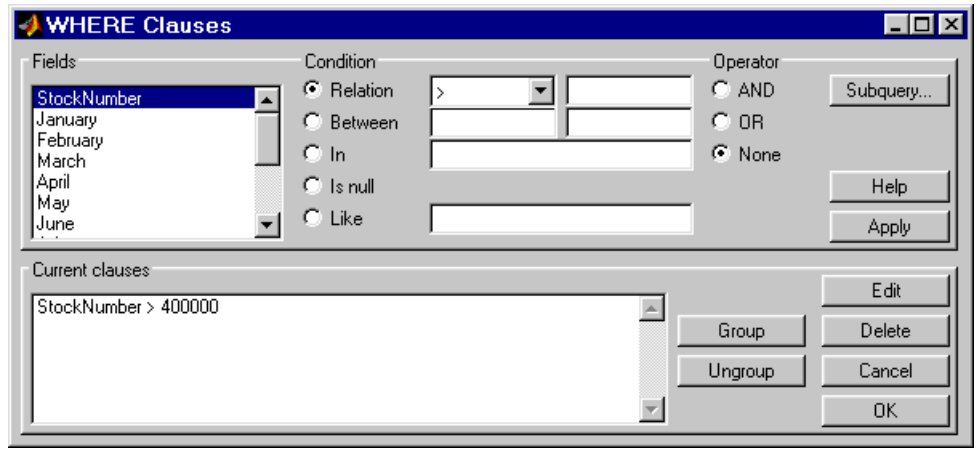
b From the drop-down list to the right of **Relation**, select >.

c In the field to the right of the drop-down list, type `400000`.



d Click **Apply**.

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



5 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
```

6 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
```

7 Click **OK**.

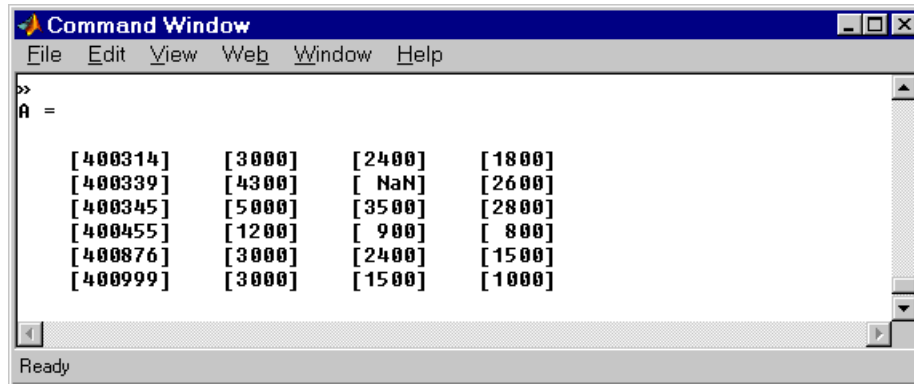
The **Where Clauses** dialog box closes. The **SQL statement** in the **Visual Query Builder** dialog box reflects the where clause you specified.

8 Assign a **MATLAB workspace variable**; for example, `A`.

9 Click **Execute**.

The results are a 6-by-4 matrix.

10 To view the results, type `A` 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 Running a Query").

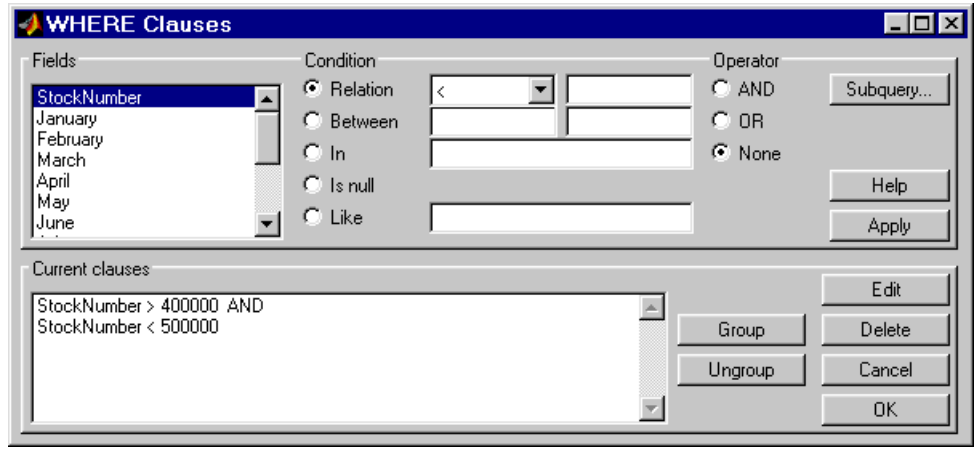


11 Select **Save** from the **Query** menu and name this query `basic_where.qry`.

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. Continuing with the example, `basic_where.qry`, where `StockNumber` is greater than `400000` and less than `50000`, modify the query to retrieve results where sales in any of the three months is greater than `1500` units, as long as sales for each of the three months is greater than `1000` units. The **Where Clauses** dialog box appears as

follows to retrieve data where the `StockNumber` is greater than `400000` and less than `50000`.



1. Add the criteria that retrieves data where sales in any of the three months is greater than 1500 units.

   a. In **Current clauses**, select `StockNumber < 500000`, and then click **Edit**.

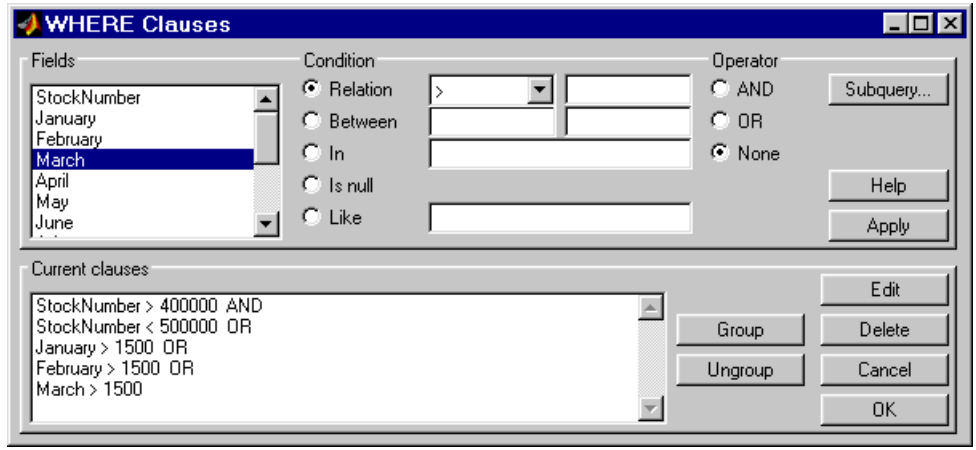
   b. For **Operator**, select `OR`, and then click **Apply**.

   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.

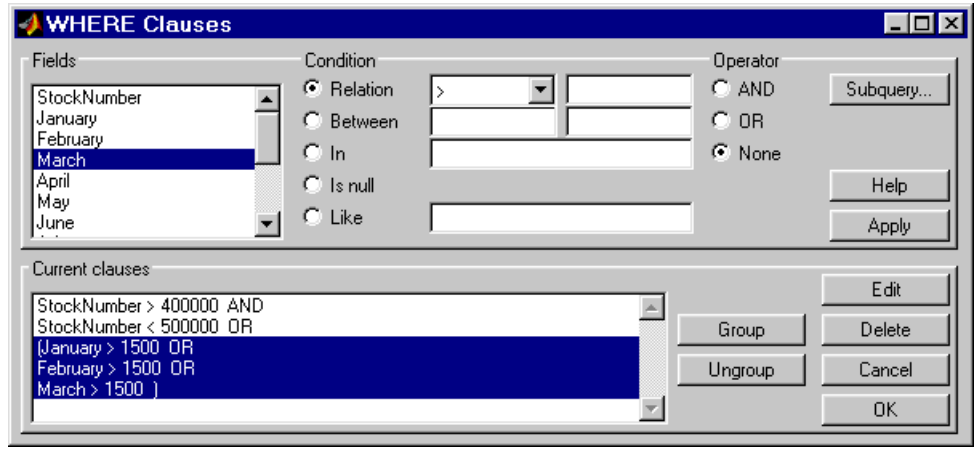
a In **Current clauses**, select the statement `January >1500 OR`.

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 three months is greater than 1000 units.

a In **Current clauses**, select the statement `March >1500)` and then click **Edit**.

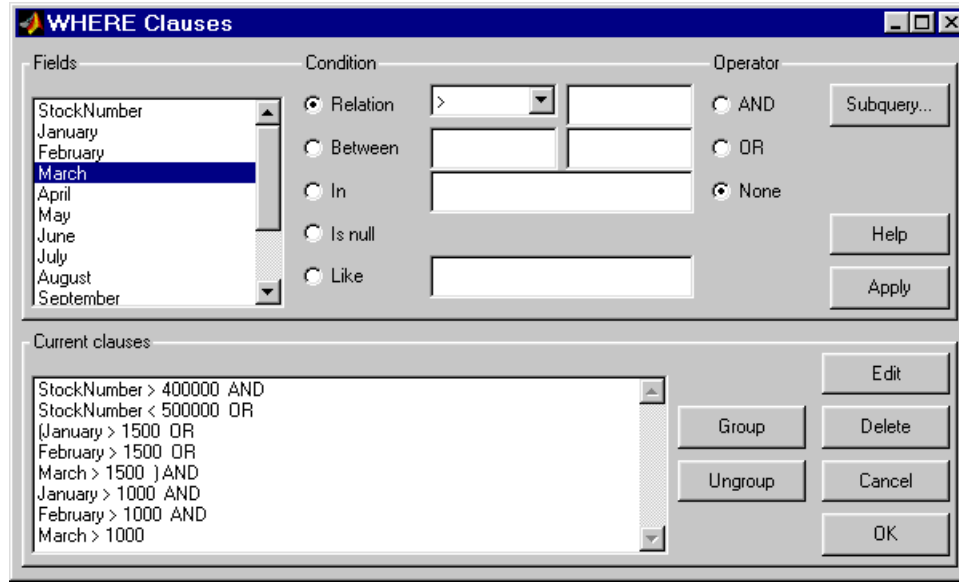
b For **Operator**, select `AND`, and then click **Apply**.

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**.

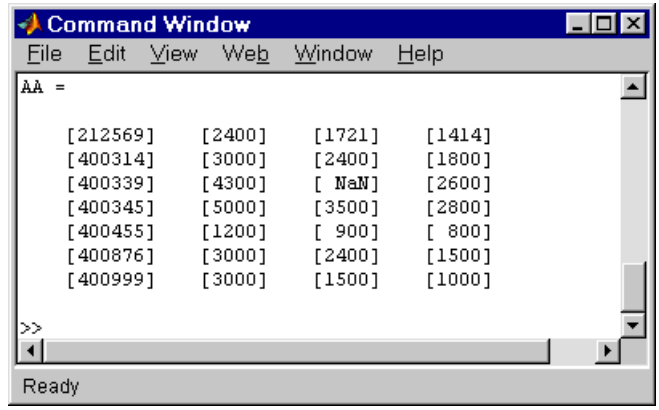
The **Where Clauses** dialog box closes. The **SQL statement** in the **Visual Query Builder** dialog box reflects the modified where clause. Because the clause is so long, you have to use the right arrow key in the field to see all of the contents.

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 clause**s, 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`, and `March > 1000`). Then click **Ungroup**. The three statements are no longer grouped.

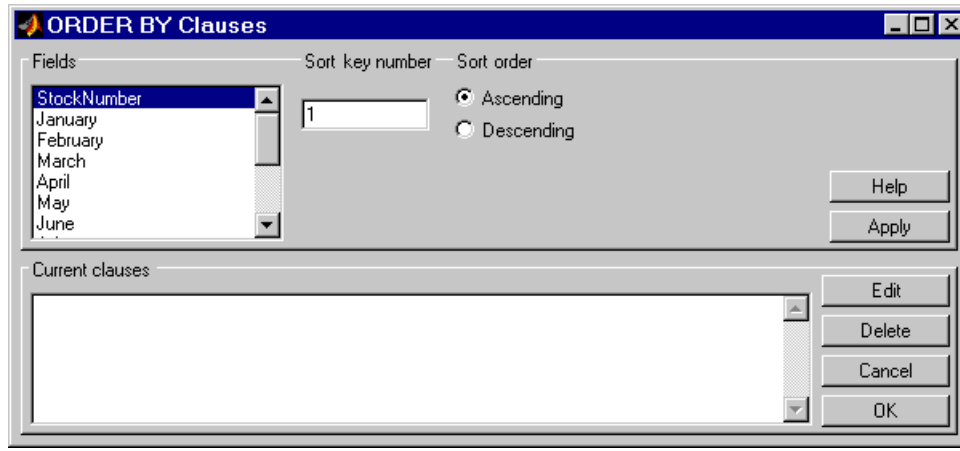
Presenting Results in Specified Order

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`, created and saved in "Retrieving Information That Meets Specified Criteria" on page 2-23. 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 for March, are descending.

1 Load `basic_where.qry`. For instructions, see "Using a Saved Query" on page 2-11.

2 In **Advanced query options**, click **Order by**.

The **ORDER BY Clauses** dialog box appears.



3 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 clause**s area now shows

```
January ASC
```

4 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 clause**s area now shows

```
January ASC
February ASC
```

5 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 clause**s area now shows

```
January ASC
February ASC
March DESC
```

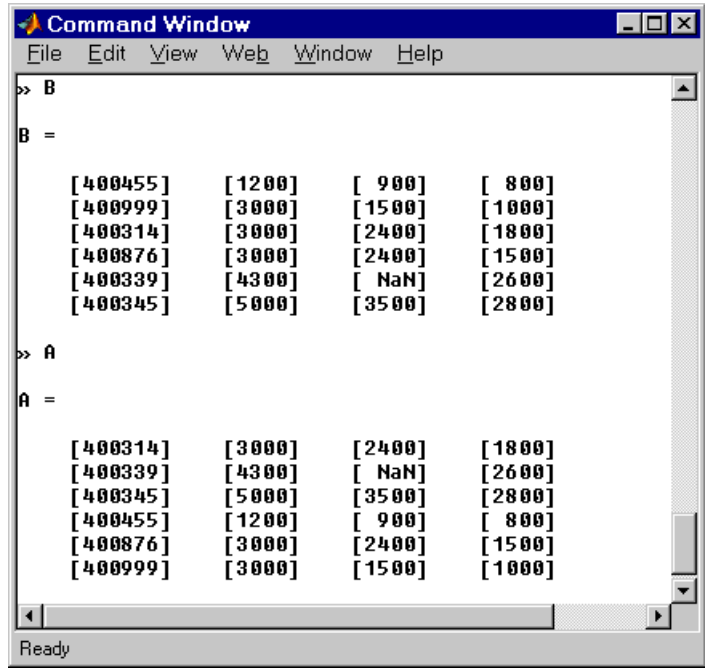
6 Click **OK**.

The **ORDER BY Clauses** dialog box closes. The **SQL statement** in the **Visual Query Builder** reflects the order by clause you specified.

7 Assign a **MATLAB workspace variable**, for example, `B`.

8 Click **Execute**.

9 To view the results, type `B` in the **Command Window**. Compare these to the unordered query results, shown as `A`.



For `B`, results are first sorted by January sales, in ascending order, from 1200 for 400455 to 5000 for 400345.

For items 400999, 400314, and 400876, January sales were equal at 3000. Therefore, the second sort key applies, February sales in ascending order, which were 1500, 2400, and 2400 respectively.

For 400314 and 400876, February sales were both 2400, so the third sort key applies, March sales in descending order, which were 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

example uses basic.qry (see "Building, Running, and Saving a Query" on page 2-7).

This example 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.

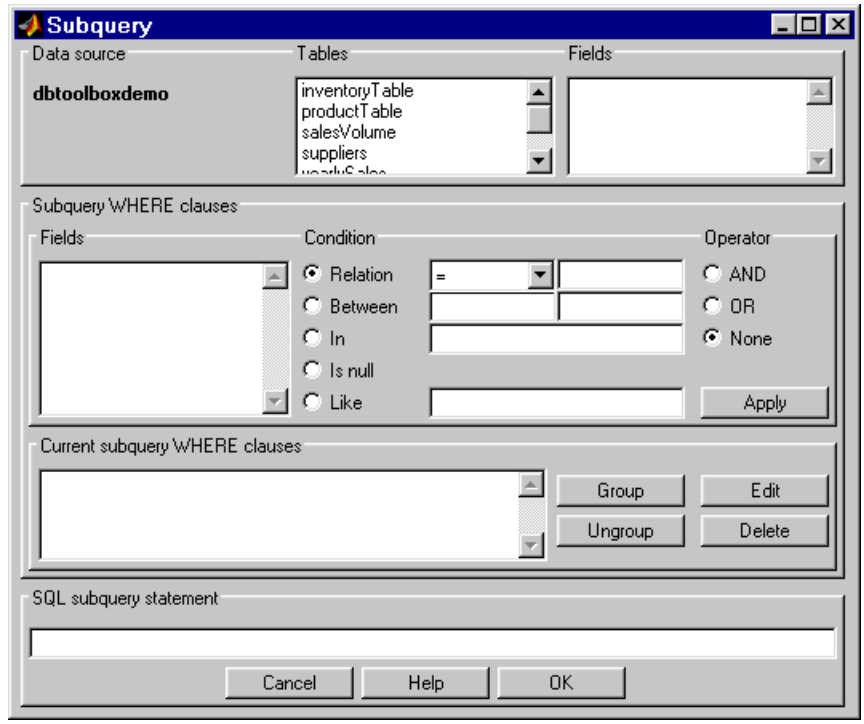
   This creates a query that retrieves the values for January, February, and March sales for all stock numbers.

2 In **Advanced query options**, click **Where**.

   The **Where Clauses** dialog box appears.

3 Click **Subquery**.

The **Subquery** dialog box appears.



4 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.

5 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`.

6 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**.

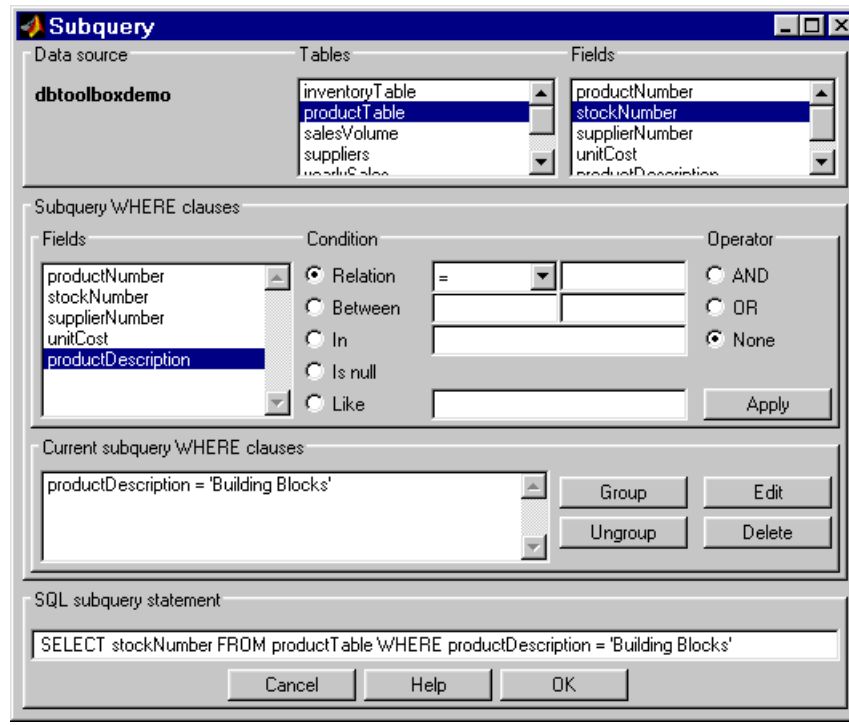
   b For **Condition**, select **Relation**.

   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).

   e Click **Apply**.

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

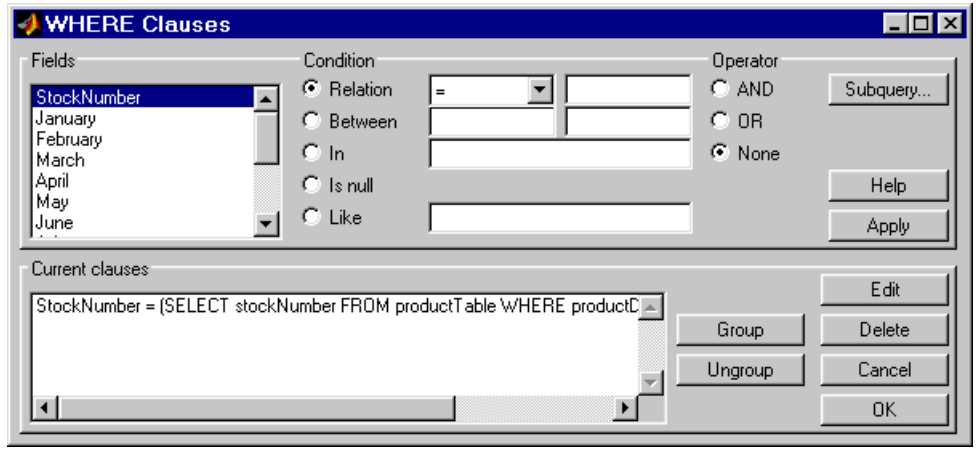


7 In the **Subquery** dialog box, click **OK**.

   The **Subquery** dialog box closes.

8 In the **WHERE Clauses** dialog box, click **Apply**.

This updates the **Current clauses** area using the subquery criteria specified in steps 2 through 7.



9 In the **WHERE Clauses** dialog box, click **OK**.

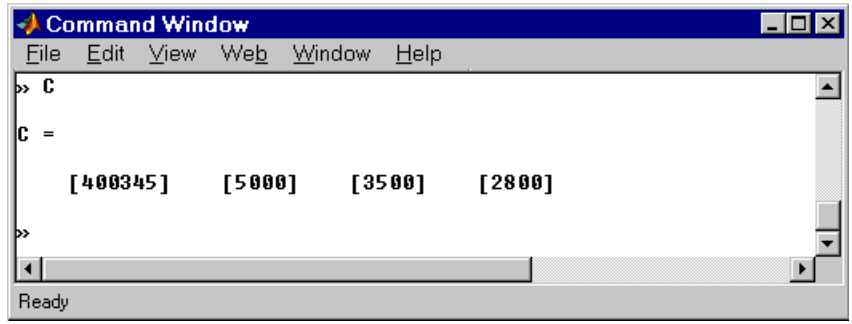
This closes the **WHERE Clauses** dialog box and updates the **SQL statement** in the **Visual Query Builder** dialog box.

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

11 Click **Execute**.

The results are a 1-by-4 matrix.

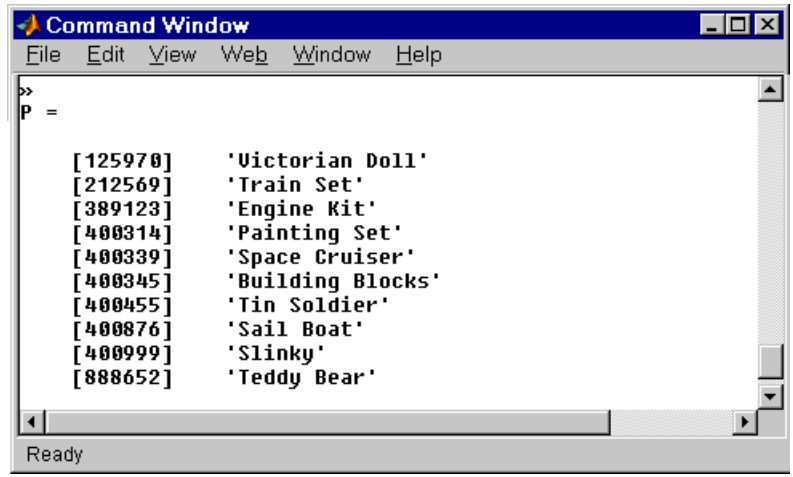
12 Type `C` at the prompt in the **MATLAB Command Window** to see the results.



13 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**.
- **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'`. “Creating Queries for Results from Multiple Tables” on page 2-41 creates a query that includes the product description in the results.

Creating Queries for Results from Multiple Tables

Select multiple tables when creating 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 2-35. 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 Select the **Data source**, for example, `dbtoolboxdemo`.

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

2 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.

3 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`.

4 In **Advanced query options**, click **Where** to make 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.

5 In the **WHERE Clauses** dialog box:

a Select `productTable.stockNumber` from **Fields**.

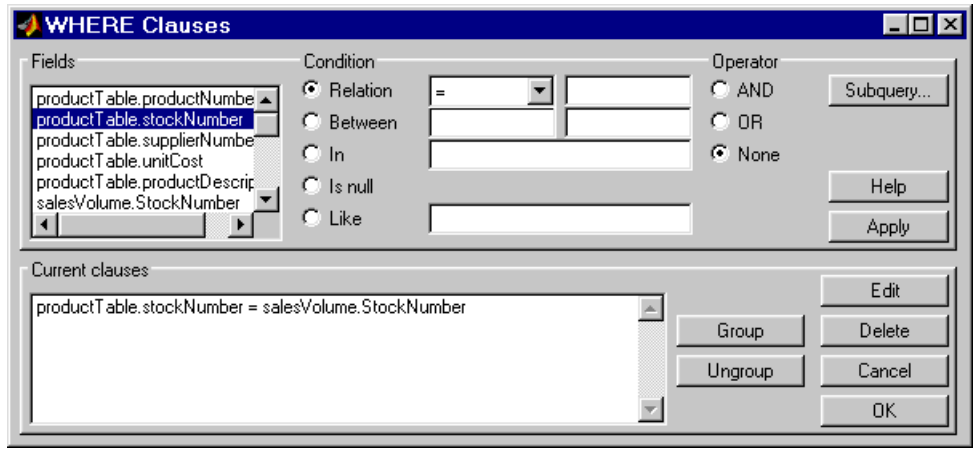
b For **Condition**, select **Relation**.

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.



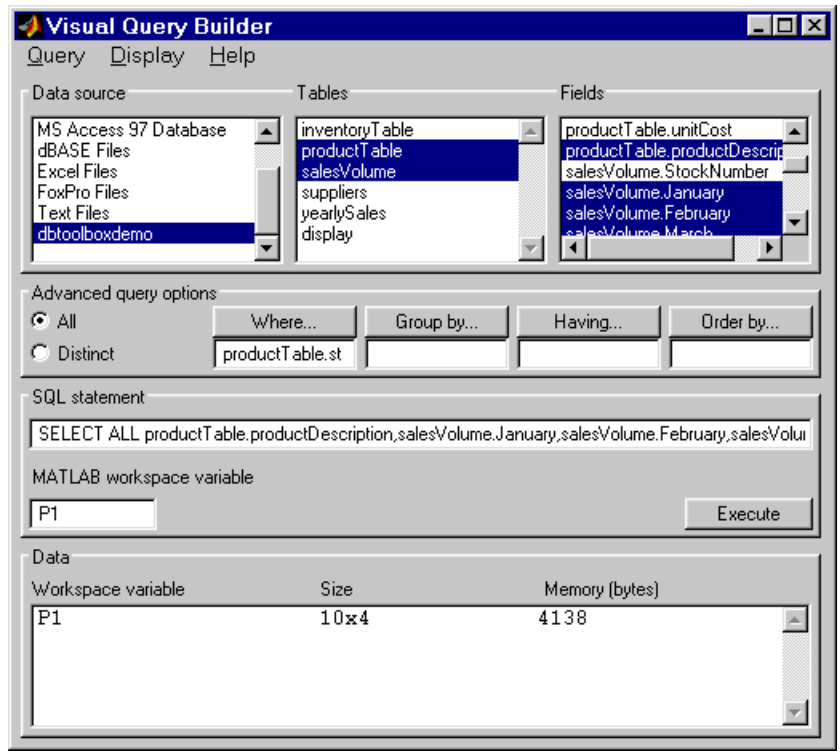
f Click **OK**.

The **WHERE Clauses** dialog box closes. The **SQL statement** in the **Visual Query Builder** dialog box reflects the where clause.

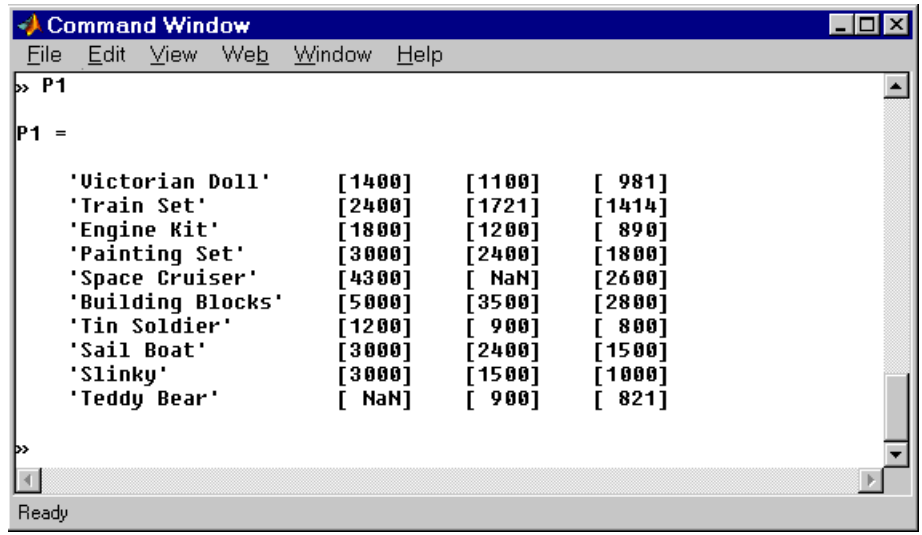
6 Assign a **MATLAB workspace variable**, for example, `P1`.

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

The results are a 10-by-4 matrix.



8 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 **Having** in **Advanced query options**, and then click **Help** in the **Having Clauses** dialog box.

3

Tutorial for Functions

Introduction

This tutorial demonstrates many of the Database Toolbox functions using simple examples.

1 "Importing Data into MATLAB from a Database" on page 3-6.

2 "Viewing Information About the Imported Data" on page 3-11.

3 "Exporting Data from MATLAB to a New Record in a Database" on page 3-14.

4 "Exporting Data from MATLAB, Replacing Existing Data in a Database" on page 3-20.

5 "Exporting Multiple Records from MATLAB" on page 3-22.

6 "Accessing Metadata" on page 3-26.

7 "Performing Driver Functions" on page 3-33.

8 "Working with Cell Arrays in MATLAB" on page 3-36.

In addition, for those who are interested in objects and methods, see "About Objects and Methods for the Database Toolbox" on page 3-4.

Examples 1 through 4 use the `SampleDB` data source. Instructions for setting up this data source are in Chapter 1, "Installation and Setup." Examples 5 and 6 use the `dbtoolboxdemo` data source. Instructions for setting up this data source are in Chapter 1, "Installation and Setup." Example 7 is not one you can run exactly as it is written since it relies on a specific JDBC connection and database, however, it serves as an illustration of what you can do. Example 8 shows some simple ways to work with cell arrays. Cell arrays are part of MATLAB's core functionality, but some users may not be familiar with them. Because the Database Toolbox makes use of cell arrays, some simple examples are included here.

If your version of Microsoft Access is different than that referred to in "Installation and Setup", 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 this tutorial.

M-files containing functions used in examples 1 through 5 are in the `matlab\toolbox\database\dbdemos` directory. As you work with the examples in this chapter, 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 this tutorial type `doc` followed by the function name, or see the “Function Reference” section.

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: [1x1
                   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."

Importing Data into MATLAB from a Database

In this part of the tutorial, 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 use these Database Toolbox functions:

- `database`
- `exec`
- `fetch`
- `logintimeout`
- `ping`

If you want to see or copy the functions for this part of the tutorial, 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".

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 – type

```
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 – type

```
ping(conn)
```

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

```
       DatabaseProductName: 'ACCESS'
```

```
    DatabaseProductVersion: '3.50.0000'
            JDBCDriverName: 'JDBC-ODBC Bridge (odbcjt32.dll)'
         JDBCDriverVersion: '1.1001 (04.00.4202)'
    MaxDatabaseConnections: 64
           CurrentUserName: 'admin'
               DatabaseURL: 'jdbc:odbc:SampleDB'
    AutoCommitTransactions: 'True'
```

5 Open a cursor and execute an SQL statement – type

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

```
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 a MATLAB cell array. Cell arrays support mixed data types.

6 Import data into MATLAB – type

```
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.

In this example, `fetch` reassigns the variable `curs` to the cursor object containing the rows of data returned by `fetch`. 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: [1x1
                   com.mathworks.toolbox.database.fetchTheData]
```

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

7 Display the Data element in the cursor object, curs. Assign the variable AA to the data element, curs.Data. Type.

```
AA = curs.Data
```

MATLAB returns

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

For more information about working with data in MATLAB cell arrays, see "Working with Cell Arrays in MATLAB" on page 3-36.

8 At this point, you can go to the next part of the tutorial. If you want to stop working on the tutorial now and resume with the next part at a later time, close the cursor and the connection. Type:

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

Viewing Information About the Imported Data

In this part of the tutorial, you view information about the data you imported and close the connection. You use these Database Toolbox functions:

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

If you want to see or copy the functions for this part of the tutorial, 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 part of the tutorial, skip this step. Otherwise, if the cursor and connection are not open, type the following to continue with this tutorial.

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

2 View the number of rows in the data set you imported – type

```
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 – type

```
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 – type

```
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 – type

```
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 – type

```
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 – type

```
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 part of the tutorial. If you want to stop working on the tutorial now and resume with the next part at a later time, close the connection. Type

```
close(conn)
```

Exporting Data from MATLAB to a New Record in a Database

In this part of the tutorial, 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, and then export the data (the average freight value and the number of shipments on which the average was based) to an empty table.

You use these Database Toolbox functions:

- `get`
- `insert`

If you want to see or copy the functions for this part of the tutorial, or if you want to run the set of functions, use the M-file `matlab\toolbox\database\dbdemos\dbinsertdemo.m`.

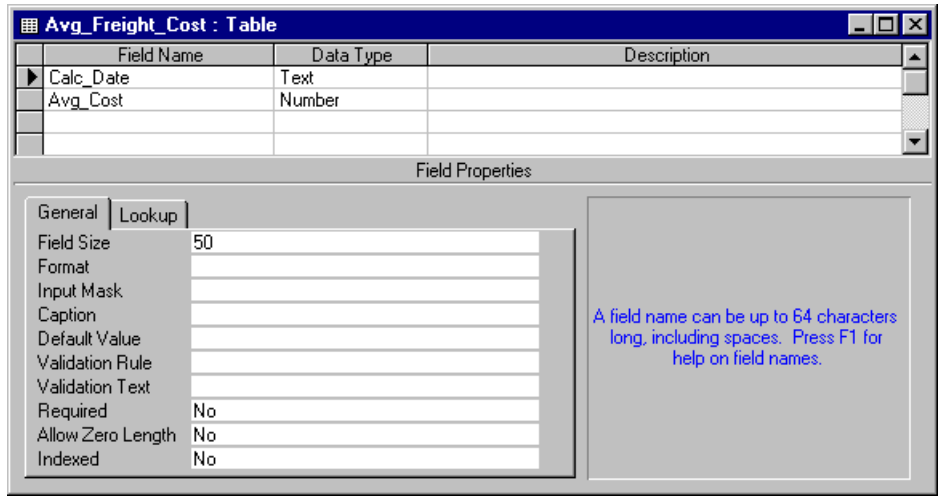
1 Create a table in Microsoft Access into which you will export MATLAB results.

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 called `Avg_Freight_Cost` 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 Close the table. Access warns you that there is no primary key, but you do not need one.

If you need more information about how to create a table in Access, see Microsoft Access help or written documentation.

Note Although Access supports the use of spaces in table and column names, most other databases do not. Therefore the Database Toolbox does not allow spaces in table and column names so do not include them. Also, be sure not to name columns using the database's reserved words, such as `DATE`, or you will not be able to import data into the database. For Access, see Access help to determine the reserved words.

---

2 If you are continuing directly from the previous part of the tutorial, skip this step. Otherwise, connect to the data source, `SampleDB`. Type

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

3 In MATLAB, import the data on which you will 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);
```

4 View the data you imported – type

```
AA = curs.Data
```

MATLAB returns

```
AA =
    [12.7500]
    [10.1900]
    [52.8400]
```

5 Calculate the average freight cost. First, assign the variable name `numrows` to the number of rows in the array. Then convert the cell array `AA` to a vector and calculate the average, assigning the result to the variable `meanA`. Divide the sum by `numrows`, but note that you must convert `numrows` to a double precision value because the divide operator, `/`, requires it. Type

```
numrows = rows(curs);
meanA = sum([AA{:}])/double(numrows)
```

MATLAB returns

```
meanA =
   25.2600
```

6 Assign the variable `D` to the date on which these orders were shipped – type

```
D = '1/20/98';
```

7 Assign the date and mean to a cell array, which will be exported to the database. Put the date in the first cell by typing

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

MATLAB returns

```
exdata =
    '1/20/98'
```

Put the mean in the second cell by typing

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

MATLAB returns

```
exdata =
 '1/20/98'    [25.2600]
```

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

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

9 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 – type

```
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.

10 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.

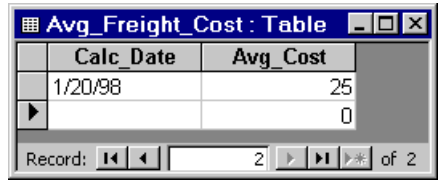
`Avg_Freight_Cost` is the name of the table to which you are exporting data. In the `insert` function, you also include the `colnames` cell array and the cell array containing the data you are exporting, `exdata`, both of which you defined in the previous steps.

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

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

```
??? Error using ==> cursor/cursor
[Microsoft][ODBC Microsoft  7.0 Driver] Table
'Avg_Freight_Cost' is exclusively locked by user '' on machine ''
```

11 In Microsoft Access, view the `Avg_Freight_Cost` table to verify the results.



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

12 Close the cursor – type

```
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.

13 At this point, you can go to the next part of the tutorial. If you want to stop working on the tutorial now and resume with the next part 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 part of the tutorial.

Exporting Data from MATLAB, Replacing Existing Data in a Database

In this part of the tutorial, you export data from MATLAB to a database, updating existing data in the database. Specifically, you update the data you previously imported into the `Avg_Freight_Cost` table.

You use these Database Toolbox functions:

- `close`
- `update`

If you want to see or copy the functions for this part of the tutorial, or if you want to run the set of functions, use the M-file `matlab\toolbox\database\dbdemos\dbupdatedemo.m`.

1 If you are continuing directly from the previous part of the tutorial, skip this step. Otherwise, type the following

```
conn = database('SampleDB', '', '');
colnames = {'Calc_Date', 'Avg_Cost'};
D = '1/20/98';
meanA = 25.2600;
exdata = {D, meanA}
```

MATLAB returns

```
exdata =
    '1/20/98'    [25.2600]
```

2 Assume that the date in the `Avg_Freight_Cost` table is incorrect and instead should be 1/19/98. Type

```
D = '1/19/98'
```

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

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

MATLAB returns

```
exdata =
    '1/19/98'    [25.2600]
```

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 `1/20/98` for the `Calc_Date`.

```
whereclause = 'where Calc_Date = ''1/20/98'''
```

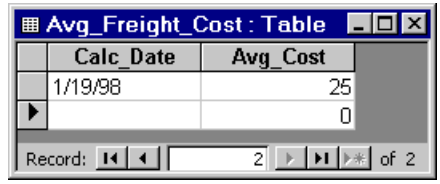
Because the date string is within a string, two single quotation marks surround the date instead of the usual single quotation mark. MATLAB returns

```
whereclause =
   where Calc_Date = '1/20/98'
```

5 Export the data, replacing the record whose `Calc_Date` is `1/20/98`.

```
update(conn, 'Avg_Freight_Cost', colnames, exdata, whereclause)
```

6 In Microsoft Access, view the `Avg_Freight_Cost` table to verify the results.



7 Disconnect from the database.

```
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 Records from MATLAB

In this example, multiple records are imported, manipulated in MATLAB, and then exported to a 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 use these Database Toolbox functions:

- `insert`
- `setdbprefs`

If you want to see or copy the functions for this part of the tutorial, or if you want to run the 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". 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 – type

   ```
   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 that any `NULL` value read from the database will be converted to a `0` in MATLAB by using the `setdbprefs` command.

   ```
   setdbprefs ('NullNumberRead','0')
   ```

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 cell array 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 – type

```
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`. To compute `monthly` using `sum`, first convert `tmp` from a cell array to a numeric array using `[tmp{:}]` because `sum` will only work on numeric arrays.

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 To export the column of data, you must first convert it to a cell array – type

```
exdata = num2cell(monthly);
```

`num2cell` takes the data in `monthly` and assigns each row to a row in a new cell array, `exdata`, which you will export in a later step.

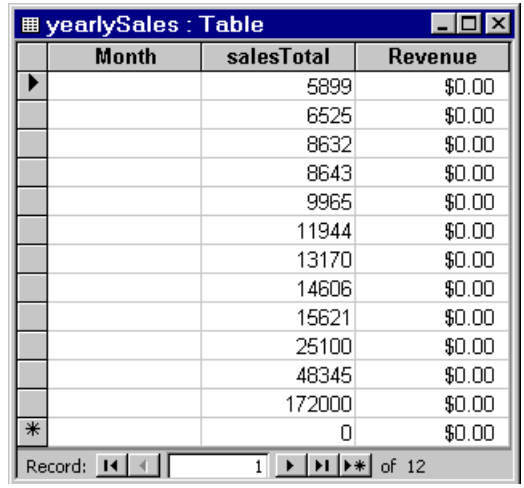
11 Create a string array containing the column names into which you are inserting the data. In a later step, we will insert the data into the `salesTotal` column of the `yearlySales` table; here we assign the variable `colnames` to the array. Type

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

12 Insert the data into the `yearlySales` table – type

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

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



14 Close the cursor and database connection. Type

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

Accessing Metadata

In this part of the tutorial, you access information about the database; this information 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` command for the metadata object you just created, `dbmeta`.

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

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

```
v =
            AllProceduresAreCallable: 1
              AllTablesAreSelectable: 1
    DataDefinitionCausesTransaction: 1
    DataDefinitionIgnoredInTransact: 0
         DoesMaxRowSizeIncludeBlobs: 0
                           Catalogs: {[1x46 char]}
                    CatalogSeparator: '.'
                         CatalogTerm: 'DATABASE'
                   DatabaseProductName: 'ACCESS'
                DatabaseProductVersion: '03.50.00'
            DefaultTransactionIsolation: 2
                     DriverMajorVersion: 1
                     DriverMinorVersion: 1001
                            DriverName: 'JDBC-ODBC Bridge
                                        (odbcjt32.dll)'
                         DriverVersion: '1.1001 (04.00.4202)'
                   ExtraNameCharacters: '~@#$%^&*_-+=\}{"';:?/><,'
                 IdentifierQuoteString: '`'
                      IsCatalogAtStart: 1
                MaxBinaryLiteralLength: 255
                  MaxCatalogNameLength: 260
                  MaxCharLiteralLength: 255
                   MaxColumnNameLength: 64
                   MaxColumnsInGroupBy: 10
                     MaxColumnsInIndex: 10
                   MaxColumnsInOrderBy: 10
                    MaxColumnsInSelect: 255
                     MaxColumnsInTable: 255
                        MaxConnections: 64
                   MaxCursorNameLength: 64
                        MaxIndexLength: 255
               MaxProcedureNameLength: 64
                           MaxRowSize: 2096
```

```
                MaxSchemaNameLength: 0
                 MaxStatementLength: 65000
                      MaxStatements: 0
                 MaxTableNameLength: 64
                  MaxTablesInSelect: 16
                  MaxUserNameLength: 0
                   NumericFunctions: [1x73 char]
                      ProcedureTerm: 'QUERY'
                            Schemas: {}
                         SchemaTerm: ''
                 SearchStringEscape: '\'
                        SQLKeywords: [1x461 char]
                   StringFunctions: [1x91 char]
     StoresLowerCaseIdentifiers: 0
StoresLowerCaseQuotedIdentifier: 0
     StoresMixedCaseIdentifiers: 0
StoresMixedCaseQuotedIdentifier: 1
     StoresUpperCaseIdentifiers: 0
StoresUpperCaseQuotedIdentifier: 0
                 SystemFunctions: ''
                      TableTypes: {4x1 cell}
                TimeDateFunctions: [1x111 char]
                        TypeInfo: {16x1 cell}
                             URL: 'jdbc:odbc:dbtoolboxdemo'
                        UserName: 'admin'
          NullPlusNonNullIsNull: 0
            NullsAreSortedAtEnd: 0
          NullsAreSortedAtStart: 0
             NullsAreSortedHigh: 0
              NullsAreSortedLow: 1
          UsesLocalFilePerTable: 0
                 UsesLocalFiles: 1
```

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 `{[1x46 char]}`. To view the actual `Catalog` information, type

```
v.Catalogs
```

MATLAB returns

```
ans =
    'D:\matlabr12\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
    DataDefinitionAndDataManipulati: 1
    DataManipulationTransactionsOnl: 0
     DifferentTableCorrelationNames: 0
```

```
                ExpressionsInOrderBy: 1
                  ExtendedSQLGrammar: 0
                      FullOuterJoins: 0
                             GroupBy: 1
                   GroupByBeyondSelect: 1
                     GroupByUnrelated: 0
          IntegrityEnhancementFacility: 0
                     LikeEscapeClause: 0
                    LimitedOuterJoins: 0
                    MinimumSQLGrammar: 1
                  MixedCaseIdentifiers: 0
           MixedCaseQuotedIdentifiers: 1
                   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 a number of 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 other Database Toolbox functions you can perform for the database metadata object, type

```
help dmd/Contents
```

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.

Performing Driver Functions

This part of the tutorial 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`

There is no equivalent M-file demo to run because the tutorial uses a PC and relies on a specific JDBC connection and database; your configuration will be different.

1 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:@144.212.33.228:1521:')
```

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

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

2 To get properties of the driver object, type

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

MATLAB returns information about the driver's versions.

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

3 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`.

4. 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.

5. Get properties of the drivermanager object. Type

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

   MATLAB returns

   ```
   v =
           Drivers: {'sun.jdbc.odbc.JdbcOdbcDriver@76163'}
      LoginTimeout: 0
         LogStream: []
   ```

6. 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: []
   ```

If you now connect to a database, the `LoginTimeout` value will be `10`. For example, type

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

MATLAB returns

```
conn =
       Instance: 'SampleDB'
       UserName: ''
         Driver: []
            URL: []
    Constructor: [1x1
                 com.mathworks.toolbox.database.databaseConnect]
        Message: []
         Handle: [1x1 sun.jdbc.odbc.JdbcOdbcConnection]
        TimeOut: 10
     AutoCommit: 'on'
           Type: 'Database Object'
```

For a list of all the driver object functions you can perform, type

```
help driver/Contents
```

Working with Cell Arrays in MATLAB

When you import data from a database into MATLAB, the data is stored in MATLAB cell arrays. You can then use MATLAB functions to work with the data. This section provides a few simple examples of how to work with cell arrays in MATLAB.

For more information on using cell arrays, see Chapter 13 of *Using MATLAB*.

Viewing Query Results

How you view query results depends on if you imported the data using the `fetch` function or if you used the Visual Query Builder.

Importing Data Using the fetch Function

If you import data from a database to MATLAB using the `fetch` function, MATLAB returns, for example

```
curs =
       Attributes: []
             Data: {3x1 cell}
   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]
```

To view the retrieved data and assign it to the workspace variable `A`, type

```
A = curs.Data
```

For this example, MATLAB returns

```
A =
    [12.7500]
    [10.1900]
    [52.8400]
```

If the query results consist of multiple columns, you can view all the results for a single column using a colon (:). For example, if running a `fetch` returns data with multiple columns, you view the results of column 2 by typing

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

MATLAB returns the data in column 2

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

Importing Data Using the Visual Query Builder

If you use the Visual Query Builder to import data, you assign the workspace variable, in this example `A`, using the Visual Query Builder and do not have to perform the above steps. Instead, just type the workspace variable name at the MATLAB prompt in the **Command Window**. For this example, type

```
A
```

MATLAB returns

```
A =
    [12.7500]
    [10.1900]
    [52.8400]
```

Viewing Results Shown as a Matrix

If the results do not fit in the limited 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)
```

MATLAB returns

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

Retrieving Elements of Query Results

For the example used in this section, the query results are assigned to the workspace variable A.

```
A =
    [12.7500]
    [10.1900]
    [52.8400]
```

Retrieving a Single Element

To retrieve a single element from A, enclose the element's row and column numbers in curly braces. For example, to retrieve the first element, type

```
A1 = A{1}
```

MATLAB returns

```
A1 = 12.75
```

Retrieving an Entire Column or Row

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

```
AA=[A{:}]
```

MATLAB returns

```
AA =
   12.7500   10.1900   52.8400
```

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

```
celldisp(A)
```

MATLAB returns

```
A{1} =
   12.7500

A{2} =
   10.1900

A{3} =
   52.8400
```

Performing Functions on Cell Arrays

To perform MATLAB functions directly on cell arrays, you need to extract the contents of the cell array by enclosing the elements in curly braces. For example, to compute the sum of the elements in the cell array `A`, type

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

Because `sum` only works on numeric arrays, you convert the contents of `A{:}` to a numeric array by enclosing it in square brackets.

Getting the Size of an Array

If you want to perform functions that use the number of rows or columns in the query results, use the `size` function to get the information. In this example, get the size of workspace variable `A`, which contains the query results, and assign the number of rows and columns in `A` to `m` and `n` respectively. Type

```
[m,n] = size(A)

m =
    10
n =
    13
```

Creating Cell Arrays for Exporting Data from MATLAB

To export data from MATLAB to a database (using the `insert` or `update` functions) you need to put the data in a cell array.

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 insert the two rows of data `A` and `avgA`, and `B` and `avgB`, use the `insert` function as follows.

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

Assigning Cell Array Elements

Put data into a cell array element by enclosing 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, by typing

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

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

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

Converting a Numeric Array to a Cell Array

To export an entire numeric array to a cell array, use the `num2cell` function. For example, to convert the numeric array `monthly` to a cell array `exdata`, type

```
exdata = num2cell(monthly);
```

`num2cell` takes the data in `monthly` and assigns each row to a row in a new cell array, `exdata`, which you can then export to your database.

4

Function Reference

Functions by Category

The following tables group Database Toolbox functions by category.

General

Function	Purpose
`logintimeout`	Set or get time allowed to establish database connection.
`setdbprefs`	Set preferences for database actions for handling `NULL` values.

Database Connection

Function	Purpose
`clearwarnings`	Clear warnings for 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.
`sql2native`	Convert JDBC SQL grammar to system's native SQL grammar.

SQL Cursor

Function	Purpose
`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

Function	Purpose
attr	Get attributes of columns in fetched data set.
cols	Get number of columns in fetched data set.
columnnames	Get names of columns in fetched data set.
fetch	Import data into MATLAB cell array.
rows	Get number of rows in fetched data set.
width	Get field size of column in fetched data set.

Exporting Data from MATLAB to a Database

Function	Purpose
commit	Make database changes permanent.
insert	Export MATLAB cell array data into database table.
rollback	Undo database changes.
update	Replace data in database table with data from MATLAB cell array.

Database Metadata Object

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

Driver Object

Function	Purpose
`driver`	Construct database driver object.
`get`	Get database driver properties.
`isdriver`	Detect if driver is a valid JDBC driver object.
`isjdbc`	Detect if driver is JDBC-compliant.
`isurl`	Detect if the database URL is valid.
`register`	Load database driver.
`unregister`	Unload database driver.

Drivermanager Object

Function	Purpose
`drivermanager`	Construct database drivermanager object.
`get`	Get database drivermanager properties.
`set`	Set database drivermanager properties.

Resultset Object

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

Resultset Metadata Object

Function	Purpose
get	Get resultset metadata properties.
rsmd	Construct resultset metadata object.

Visual Query Builder

Function	Purpose
confds	Configure data source for use with Visual Query Builder (JDBC only).
querybuilder	Start visual SQL query builder.

Alphabetical List of Functions

This section contains detailed descriptions of all Database Toolbox functions. You can also access this information through the doc function, or the Help browser feature for searching by function name.

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
`fieldName`	Name of the column
`typeName`	Data type
`typeValue`	Numerical representation of the data type
`columnWidth`	Size of the field
`precision`	Precision value for floating and double data types; an empty value is returned for strings
`scale`	Precision value for real and numeric data types; an empty value is returned for strings
`currency`	If `true`, data format is currency
`readOnly`	If `true`, the data cannot be overwritten
`nullable`	If `true`, the data can be `NULL`
`Message`	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`

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`, is
- `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`

clearwarnings

Purpose

Clear warnings for database connection or resultset

Syntax

```
clearwarnings(conn)
clearwarnings(rset)
```

Description

`clearwarnings(conn)` clears the warnings reported for the database connection object `conn`, which was created using `database`.

`clearwarnings(rset)` clears the warnings reported for the resultset object `rset`, which was created using `resultset`.

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

```
help database/clearwarnings
help resultset/clearwarnings
```

Examples

`clearwarnings(conn)` `NULLS` reported warnings for the database connection object `conn`, which was created using `conn = database(...)`.

See Also

`database`, `get`, `resultset`

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`.

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

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
```

close

Examples

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

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

See Also

`database`, `exec`, `fetch`, `resultset`

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`

columnnames

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`

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 3rd 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`

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

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`

confds

Purpose

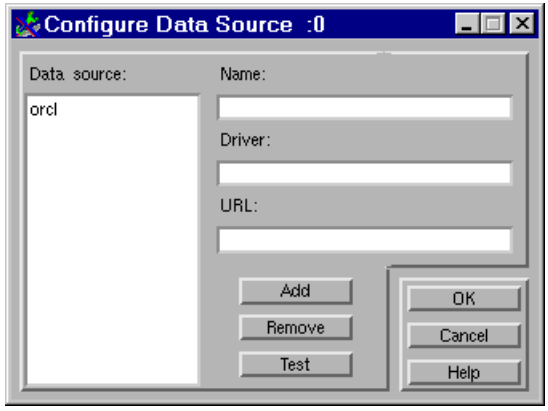
Configure data source for use with Visual Query Builder (JDBC only)

Syntax

```
confds
```

Description

`confds` displays the **Configure Data Source** dialog box, from which you add and remove data sources. Use `confds` if you connect to databases via JDBC drivers and want to use the Visual Query Builder. To add and remove data sources for connections that use ODBC drivers, see "Setting Up a Data Source" in Chapter 1 of the *Database Toolbox User's Guide*.



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

   Name: `orcl`

   Driver: `oracle.jdbc.driver.OracleDriver`

   URL: `jdbc:oracle:thin:@144.212.33.130:1521:`
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**.

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.

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`

database

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 JDBC driver name is oracle.jdbc.driver.OracleDriver and the URL to the database is jdbc:oracle:oci7:.

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

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:@144.212.33.130:1521:')` 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.

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.

After opening a cursor, use `fetch` to import data from the cursor. Use `resultset`, `rsmd`, and `statement` to get properties of the cursor.

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 `conn`. Assign the variable `curs` to the returned cursor object.

```
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 `conn`. Assign the variable `sqlquery` to the SQL statement and assign `curs` to the returned cursor.

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

Example 3 – 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 `conn`, type

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

To commit the data, type:

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

Example 4 – Run Stored Procedure

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

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

See Also

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

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.

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`

fetch

Purpose

Import data into MATLAB cell array

Syntax

```
curs = fetch(curs, RowLimit)
curs = fetch(curs)
curs.Data
```

Description

`curs = fetch(curs, RowLimit)` imports rows of data from the open SQL cursor `curs`, up to the specified `RowLimit`, into the object `curs`. It is common practice to reassign the variable `curs` from the open SQL cursor to the object returned by `fetch`. The next time you run `fetch`, records are imported starting with the row following `RowLimit`.

`curs = fetch(curs)` imports rows of data from the open SQL cursor `curs`, up to the `RowLimit` specified by `set`, into the object `curs`. It is common practice to reassign the variable `curs` from the open SQL cursor to the object returned by `fetch`. The next time you run `fetch`, records are imported starting with the row following `RowLimit`. If no `RowLimit` was specified by `set`, `fetch` imports all remaining rows of data.

Running `fetch` returns information about the cursor object. The `Data` element of the cursor object points to the cell array that contains the data returned by `fetch`. The data types are preserved (cell arrays support mixed data types). After running `fetch`, display the returned data by typing `curs.Data`.

Use `get` to view properties of `curs`.

Examples

Example 1 – Import All Rows of Data

Import all of the data into the cursor object `curs`.

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

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 pointed to by the element `curs.Data` of the cursor object. To display data in the cell array `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'
```

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.

Property	Value
`'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.

'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 `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
```

MATLAB returns the catalog names

```
ans =
    'D:\matlabr12\toolbox\database\dbdemos\db1'
    'D:\matlabr12\toolbox\database\dbdemos\origtutorial'
    'D:\matlabr12\toolbox\database\dbdemos\tutorial'
    'D:\matlabr12\toolbox\database\dbdemos\tutorial1'
```

See Also

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

importedkeys

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`

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

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`

Purpose

Export MATLAB cell array data into database table

Syntax

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

Description

`insert(conn, 'table', colnames, exdata)` exports records from the MATLAB cell array `exdata`, into new rows in an existing database table `tab`, via the connection `conn`. Specify the column names for `tab` as strings in the MATLAB cell array, `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.

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

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

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

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

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

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

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

To export the data, it must be in a cell array. The variable `sale_price` is a numeric array because it was the result of the discount calculation. You must convert `sale_price` to a cell array. To convert the columns of data in `sale_price` to a cell arrays, type

```
sale_price = num2cell(sale_price);
```

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 `sale_price` in column three.

```
exdata = id(:,1);
exdata(:,2) = price;
exdata(:,3) = sale_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 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
```

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:@144.212.33.130:1521:', d)
```

and MATLAB returns

```
a =
     1
```

indicating that the database URL, `jdbc:odbc:thin:@144.212.33.130:1521:`, 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 timeout 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 timeout 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.

Examples

Example 1 – Get Timeout Value for ODBC Connection

Your database connection is via an ODBC connection. To see the current timeout value, type

```
logintimeout
```

MATLAB returns

```
ans =
     0
```

The timeout value has not been set.

Example 2 – Set Timeout Value for ODBC Connection

Set the timeout value to five seconds for an ODBC driver. Type

```
logintimeout(5)
```

MATLAB returns

```
ans =
     5
```

Example 3 – Get and Set Timeout Value for JDBC Connection

Your database connection is via the Oracle JDBC driver. First see what the current timeout value is. Type

```
logintimeout('oracle.jdbc.driver.OracleDriver')
```

MATLAB returns

```
ans =
     0
```

The timeout value is currently `0`. Set the timeout to 10 seconds. Type

```
timeout = logintimeout('oracle.jdbc.driver.OracleDriver', 10)
```

MATLAB returns

```
timeout =
     10
```

Verify the timeout value for the JDBC driver. Type

```
logintimeout('oracle.jdbc.driver.OracleDriver')
```

MATAB 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:@144.212.33.
                            228:1521:orcl'
    AutoCommitTransactions: 'True'
```

Example 3 – Unsuccessful Request for Information About Connection

Type

```
ping(conn)
```

where 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`

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'
```

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

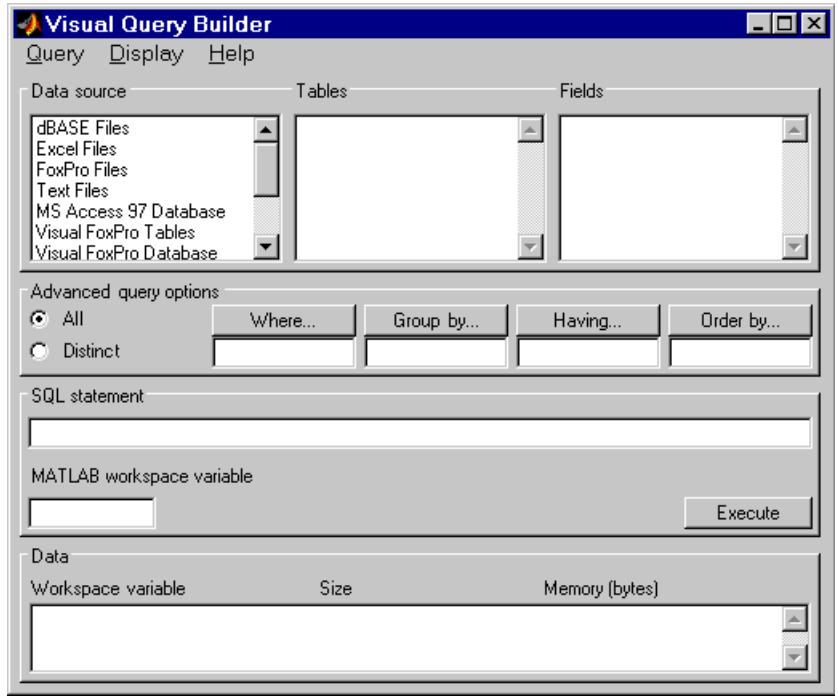
Start visual SQL query builder

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 Chapter 2, “Visual Query Builder Tutorial”. 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 timeout value is defined for a database by the database administrator. If the timeout value is zero, a query must be completed immediately.

Examples

Get the current database timeout setting for `curs`.

```
querytimeout(curs)

ans =
     10
```

Limitations

If a database does not have a database timeout 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`

Purpose

Undo database changes

Syntax

```
rollback(conn)
```

Description

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

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)
```

Roll back the data inserted in the database by typing

```
rollback(conn)
```

The data in `exdata` is removed from the database so the database contains the same data it did before the `insert`.

See Also

`commit`, `database`, `exec`, `get`, `insert`, `update`

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`

Purpose

Construct resultset metadata object

Syntax

```
rsmeta = rsmd(rset)
rsmeta = rsmd(curs)
```

Description

`rsmeta = rsmd(rset)` creates a resultset metadata object `rsmeta`, for the resultset object `rset`, or the cursor object `curs`, where `rset` was created using `resultset`, and `curs` was created using `exec` or `fetch`. Get properties of `rsmeta` using `get`, or make calls to `rsmeta` using your own Java-based applications.

Examples

Type

```
rsmeta=rsmd(rset)
```

MATLAB returns

```
rsmeta =
     Handle: [1x1 sun.jdbc.odbc.JdbcOdbcResultSetMetaData]
```

Use `v = get(rsmeta)` and `v.property` to see properties of the resultset metadata object.

See Also

`exec`, `get`, `resultset`

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`	*Not* 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.

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:@144.212.33.228:1521:');
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

Purpose

Sets preferences for database actions for handling NULL values

Syntax

```
setdbprefs
setdbprefs('property')
setdbprefs('property', 'value')
setdbprefs({'property1'; ... ;'propertyn'}, {'value1'; ... ;
  'valuen'})
```

Description

`setdbprefs` returns the current values for database action preferences.

`setdbprefs('property')` returns the current preference value for the specified `property`.

`setdbprefs('property', 'value')` sets the preference to `value` for the specified `property`.

`setdbprefs({'property1'; ... ;'propertyn'}, {'value1'; ... ; 'valuen'})` sets the preference values to `value1` through `valuen` for the properties `property1` through `propertyn`.

Allowable properties are listed in the following table.

Allowable Properties	Description
`'NullNumberRead'`	How `NULL` numbers in a database are represented when imported into MATLAB
`'NullNumberWrite'`	Numbers in MATLAB that are represented as `NULL` when exported to a database
`'NullStringRead'`	How `NULL` strings in a database are represented when imported into MATLAB
`'NullStringWrite'`	Strings in MATLAB that are represented as `NULL` when exported to a database

Examples

Example 1 – `setdbprefs`

Type `setdbprefs` and MATLAB returns

```
NullNumberRead: 'NaN'
```

```
      NullNumberWrite: 'NaN'
       NullStringRead: 'null'
      NullStringWrite: 'null'
```

which means:

- 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 – `setdbprefs(property)`

Type `setdbprefs ('NullNumberRead')` and MATLAB returns

```
NullNumberRead: '0'
```

which means any `NULL` number in the database is read into MATLAB as `0`.

Example 3 – `setdbprefs(property, value)`

Type `setdbprefs ('NullStringWrite','NaN')`

which means that any `'NaN'` string in MATLAB is exported to the database as a `NULL` string.

Example 4 – `setdbprefs({'property1'; ... ;'propertyn'}, ... {'value1';'valuen'})`

Type

```
setdbprefs({'NullStringRead';'NullStringWrite';...
'NullNumberRead';'NullNumberWrite'},{'null';'null';'NaN';'NaN'})
```

which means:

- 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
- 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

sql2native

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` from JDBC SQL grammar into the database system's native SQL grammar, returning the native SQL statement to `n`.

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`

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`

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`

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`

update

Purpose

Replace data in database table with data from MATLAB cell array

Syntax

```
update(conn, 'tab', colnames, exdata, 'whereclause')
```

Description

`update(conn, 'tab', colnames, exdata, 'whereclause')` exports data from the MATLAB cell array `exdata`, into the database table `tab`, via the database connection `conn`. It replaces existing records in the table as specified by the SQL command `whereclause`. Specify the column names for `tab` as strings in the MATLAB cell array, `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`

versioncolumns

Purpose

Get automatically updated table columns

Syntax

```
vl = versioncolumns(dbmeta, 'cata')
vl = versioncolumns(dbmeta, 'cata', 'sch')
vl = versioncolumns(dbmeta, 'cata', 'sch', 'tab')
```

Description

`vl = 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`.

`vl = 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`.

`vl = 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

```
vl = versioncolumns(dbmeta,'orcl','SCOTT','BONUS','SAL')
```

MATLAB returns

```
vl =
     {}
```

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.

See Also `columns`, `dmd`, `get`

width

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`

Index

D

E

F

G

H

I

P

Q

R

S

T

U

V

W