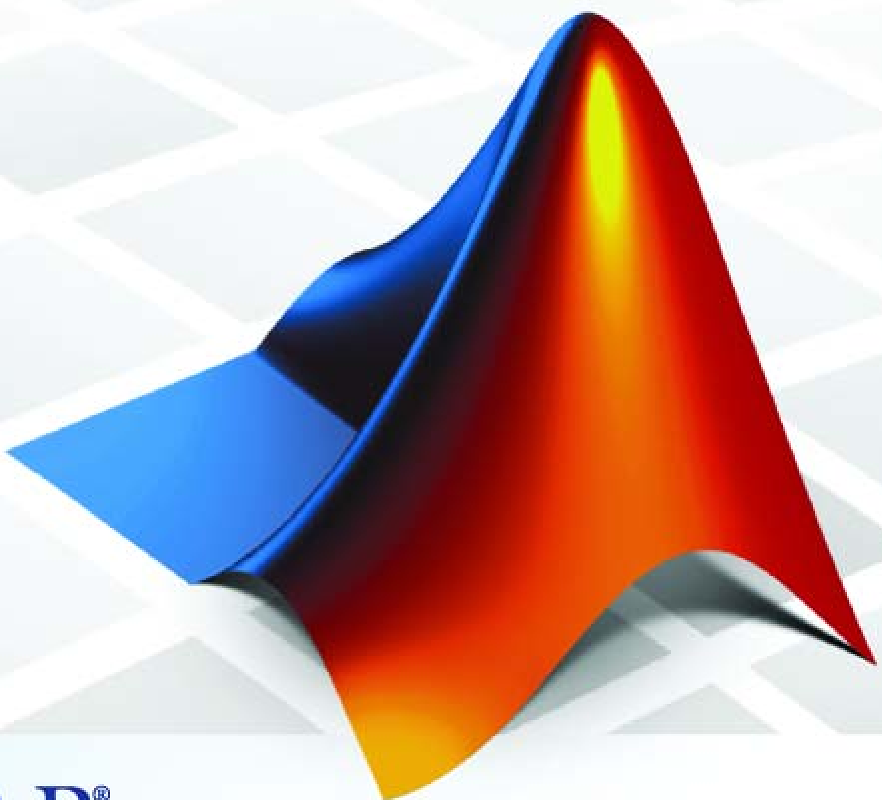
Database Toolbox 3
User's Guide



MATLAB®

The MathWorks
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Database Toolbox User's Guide

Trademarks

Patents

Revision History

May 1998	Online Only	New for Version 1 for MATLAB 5.2
July 1998	First Printing	For Version 1
Online only	June 1999	Updated for Version 2 (Release 11)
December 1999	Second printing	For Version 2 (Release 11)
Online only	September 2000	Updated for Version 2.1 (Release 12)
June 2001	Third printing	Updated for Version 2.2 (Release 12.1)
July 2002	Online only	Updated for Version 2.2.1 (Release 13)
November 2002	Fourth printing	Version 2.2.1
June 2004	Fifth printing	Updated for Version 3.0 (Release 14)
October 2004	Online only	Updated for Version 3.1 (Release 14SP1)
March 2005	Online only	Updated for Version 3.0.2 (Release 14SP2)
September 2005	Online only	Updated for Version 3.1 (Release 14SP3)
March 2006	Online only	Updated for Version 3.1.1 (Release 2006a)
September 2006	Online only	Updated for Version 3.2 (Release 2006b)
October 2006	Sixth printing	Updated for Version 3.2 (Release 2006b)
March 2007	Online only	Updated for Version 3.3 (Release 2007a)

Contents

Getting Started

What Is Database Toolbox?

This section covers the following topics:

Overview

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

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

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

How Database Toolbox Works with Databases

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

Visual Query Builder (VQB), which comes with Database Toolbox, is an easy-to-use graphical user interface (GUI) for exchanging data with your database. You can use VQB instead of or in addition to using Database

Toolbox functions. With VQB, you retrieve data by selecting information from lists to build queries. VQB imports the data into the MATLAB workspace so you then can process the data using the MATLAB suite of functions. With VQB, you can display the retrieved information in relational tables, reports, and charts. You can also use VQB to export data from MATLAB and insert it into new rows in a database.

Features of Database Toolbox

Database Toolbox has the following features:

- Different databases can be used in a single session — Import data from one database, perform calculations, and export the modified or unmodified data to another database. Multiple databases can be open during a session.
- Data types are automatically preserved in MATLAB — No data massaging or manipulation is required. The data is stored in MATLAB as cell arrays or structures, which support mixed data types, or as numeric matrices, per your specification. Export numeric, cell array, or structure data.
- Retrieval of large data sets or partial data sets — Import large data sets from a database in a single fetch or in discrete amounts using multiple fetches.
- Retrieval of `BINARY` or `OTHER` JDBC data types — You can import and export Java objects such as bitmap images.
- Retrieval of database metadata — You do not need to know table names, field names, and properties of the database structure to access the database, but can retrieve that information using Database Toolbox metadata functions.
- Dynamic importing of data from within MATLAB — Modify your SQL queries in MATLAB statements to retrieve the data you need.
- Single environment for faster data analysis — Access both database data and MATLAB functions at the MATLAB command prompt.
- Multiple cursors supported for a single database connection — Once you establish a connection with a database, the connection can support the use of multiple cursors. You can execute several queries on the same connection.

- Export query results using Report Generator — If Report Generator product is installed locally, you can create custom reports from Visual Query Builder.

- Database connections remain open until explicitly closed — Once you establish the connection to a database, 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 and export data.
- Visual Query Builder — Exchange information with databases via this easy-to-use graphical interface (GUI), even if you are unfamiliar with SQL.

Note Perform database administrative tasks, such as creating tables, using your database system application. Database Toolbox is not intended to be used as a tool for database administration.

Expected Background for Users

MATLAB

This documentation assumes you have a basic working understanding of MATLAB. You need to know about working with cell arrays and structures.

Database Connection

To connect to a database with Database Toolbox, you need to know where your data source and database driver are located. If you do not have that information, consult your database administrator when you perform the instructions for setting up a data source.

SQL

It is not required that you be familiar with Structured Query Language (SQL) to use Database Toolbox. If you are not familiar with SQL and database applications, use the Visual Query Builder (VQB) tool.

If you are familiar with SQL and the database applications you use, you can use VQB and Database Toolbox functions.

You should be familiar with SQL to perform complex queries and database operations.

Using Visual Query Builder Versus Functions

These guidelines describe the main differences between Visual Query Builder and Database Toolbox functions.

When to Use Visual Query Builder

Use Visual Query Builder to

- Retrieve data from relational databases for use in MATLAB when you are not familiar with the Structured Query Language (SQL).
- Insert data from MATLAB into new records in a database when you are not familiar with SQL.
- Easily build SQL queries and exchange data between databases and MATLAB using a GUI.
- View the SQL statement for queries you generate with VQB, and directly edit the statements.
- Automatically generate a MATLAB M-file that consists of Database Toolbox functions to perform the query you built using VQB.

When to Use Database Toolbox Functions

You can use Database Toolbox functions to do everything VQB does and more. Database Toolbox functions offer these additional features:

- Replace data in databases from MATLAB.
- Write MATLAB M-files and applications that access databases.
- Use the `fastinsert` function to export binary data or other data types that you can import but cannot export with VQB.
- Export data more quickly using the `fastinsert` function.
- Perform other functions not available with Visual Query Builder, for example, accessing database metadata.

System Requirements

Database Toolbox works with the systems and applications described here:

- “Platforms” on page 1-7
- “MATLAB and Related Products” on page 1-7
- “Databases” on page 1-8
- “Drivers” on page 1-9
- “Structured Query Language (SQL)” on page 1-10
- “Data Types” on page 1-11

Platforms

Database Toolbox runs on all of the platforms that support MATLAB, but you cannot run MATLAB with the `-nojvm` startup option.

MATLAB and Related Products

Database Toolbox requires MATLAB. To use Visual Query Builder feature for creating customized reports in Report Generator, you need the MathWorks Report Generator product. Without that product you can use VQB’s similar **Display > Report**.

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with Database Toolbox. See more information about these products on the MathWorks Web site, at `http://www.mathworks.com/products/database/related.jsp`.

Databases

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

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

RDBMS for VAX/VMS systems is not supported.

If you are upgrading from an earlier version of a database, such as Microsoft SQL Server 2000, to a newer version, there is nothing special you need to do for Database Toolbox. Just configure the data sources for the new version of the database application as you did for the original version.

Data Retrieval Restrictions

Spaces in Table and Column Names. Microsoft Access supports the use of spaces in table and column names, but most other databases do not. When using functions to retrieve data from tables and fields whose names contain spaces, use delimiters around the table and field names to build the query. For Access, enclose the table or field name in quotation marks, for example, `"order id"`. Other databases use different delimiters, for example, brackets, `[ ]`, instead of quotation marks. In Visual Query Builder, table and field names that include spaces appear in quotation marks.

Quotation Marks in Table and Column Names. Do not include quotation marks in table and column names. Database Toolbox does not support data retrieval from table and column names that contain quotation marks.

Reserved Words in Column Names. Be sure not to name columns using the database's reserved words, such as `DATE` in Microsoft Access, or you will not be able to import or export the data using Database Toolbox. You will get an error message in the MATLAB Command Window, for example, reporting a syntax error from Microsoft Access.

Drivers

For Windows platforms, Database Toolbox supports Open Database Connectivity (ODBC) drivers as well as Java Database Connectivity (JDBC) drivers.

For UNIX platforms, Database Toolbox supports Java Database Connectivity (JDBC) drivers.

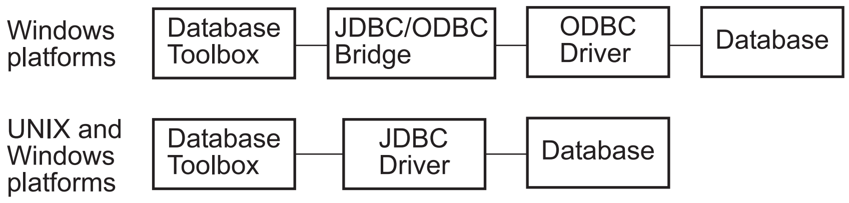
The driver for your database must be installed in order to use Database Toolbox. Most users (or their database administrators) install the driver when they install the database. Consult your database documentation or your database administrator if you need instructions to install a database driver. If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

About Drivers for Database Toolbox

An ODBC driver is a standard Windows 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.

Database Toolbox is a Java-based application. To connect 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 MATLAB JVM.

The following illustrates the use of drivers with Database Toolbox.



If your Windows-based database supports both ODBC and JDBC drivers, the JDBC drivers might provide better performance when you access the database because the ODBC/JDBC bridge is not used.

Structured Query Language (SQL)

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

Data Types

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

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

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

Using the `fastinsert` function (instead of the `insert` function or the VQB insert feature), you can export any type of data that you can import with Database Toolbox.

BINARY and OTHER Java SQL Data Types

You can import `BINARY` (referred to as BLOB for Binary Large Objects) and `OTHER` JDBC objects. To use these data types in MATLAB, you need to understand the content, and you might need to adjust it, such as stripping off headers created by the ODBC/JDBC drivers, so a specific binary format can be used in MATLAB. You can export binary data using `fastinsert`.

For an example using Visual Query Builder, see "Retrieving BINARY and OTHER Java Data" on page 2-57. For an example using functions, see "Retrieving BINARY or OTHER Java SQL Data Types" on page 3-25. In some cases, retrieving `OTHER` data types does not result in any data.

Setting Up a Data Source

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

Instructions for setting up a data source depend on the type of database driver, ODBC or JDBC:

- ODBC — For MATLAB Windows platforms only, whose database resides on a PC or on another system to which the PC is networked via ODBC drivers. See "Setting Up a Data Source for ODBC Drivers" on page 1-13.
- JDBC — For MATLAB platforms that connect to a database via a JDBC driver. See "Setting Up a Data Source for JDBC Drivers" on page 1-20.

For background information, see "Drivers" on page 1-9.

Setting Up a Data Source for ODBC Drivers

This procedure is to set up a data source for a PC running Windows whose database resides on that PC or on another system to which the PC is networked via an ODBC driver.

- Prepare examples: The examples in this documentation are based on Microsoft Access. If you have Microsoft Access installed and want to follow along with the examples, first get the databases and prepare them—see "Prepare to Use Examples" on page 1-14.
- Define the data source: To define the data source after preparing to use the examples, or to define any ODBC data source, see "Define an ODBC Data Source" on page 1-16.

Prepare to Use Examples

Prepare to use the following data sources so that you can follow the examples in this documentation:

- "dbtoolboxdemo Data Source" on page 1-14
- "SampleDB Data Source" on page 1-14

dbtoolboxdemo Data Source. The `dbtoolboxdemo` data source uses the `tutorial` database provided with Database Toolbox, *`matlabroot`*`/toolbox/database/dbdemos/tutorial.mdb`. The *`matlabroot`* directory is where MATLAB is installed on your system, as determined by running the `matlabroot` function in the Command Window. Before you can define this data source, perform the following actions:

1. Using operating system features or the MATLAB `copyfile` function, copy `tutorial.mdb` to a different directory for which you have write access, and rename it to `tutorial_copy.mdb`.
2. Using operating system features or the MATLAB `fileattrib` function, ensure you have write access to `tutorial_copy.mdb`.
3. Open `tutorial_copy.mdb` in Access. One way to do this from within the MATLAB Current Directory browser: select the file and select **Open Outside MATLAB** from the context menu. You might have to convert the database to your version of Access. Save the database as `tutorial.mdb`.

SampleDB Data Source. The `SampleDB` data source uses the Microsoft Access sample database called `Nwind`.

1. If you do not already have the `Nwind` database on your system, you can download it from the Microsoft Web site downloads page. The version referred to in this documentation is part of the Access 2000 downloads and is the Northwind Traders sample database, `Nwind.exe`. Run the file to create the `Nwind.mdb` database.
2. Using operating system features or the MATLAB `fileattrib` function, ensure you have write access to `Nwind.mdb`.
3. Rename the file to `Nwind_orig.mdb`.

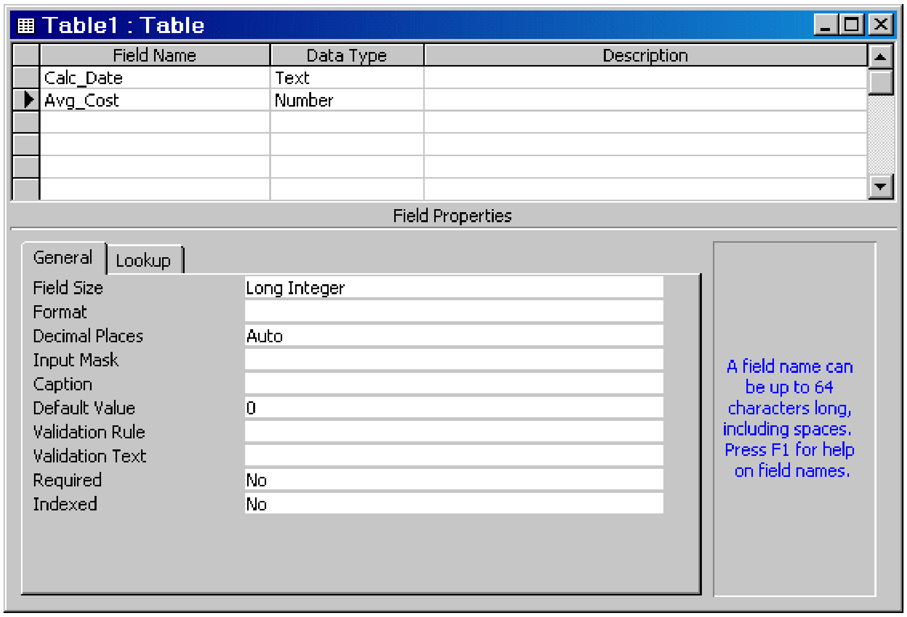
4 Open Nwind_orig.mdb in Access. From within the MATLAB Current Directory browser, you can do this by selecting the file and selecting **Open Outside MATLAB** from the context menu. You might have to convert the database to your version of Access. Save the database as Nwind.mdb.

5 Using Access, create a table into which you will export MATLAB results:

a Open the Nwind database in Microsoft Access.

b Create a new table that has two columns, Calc_Date and Avg_Cost.

c 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.



Field Name	Data Type	Description
Calc_Date	Text	
Avg_Cost	Number	

Field Size	Long Integer
Format	
Decimal Places	Auto
Input Mask	
Caption	
Default Value	0
Validation Rule	
Validation Text	
Required	No
Indexed	No

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

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

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

Define an ODBC Data Source

These instructions for defining the ODBC data source use as an example the Microsoft ODBC driver Version 4.00 and the U.S. English version of Microsoft Access 2000 for Windows. If you have a different configuration, you may have to modify the instructions.

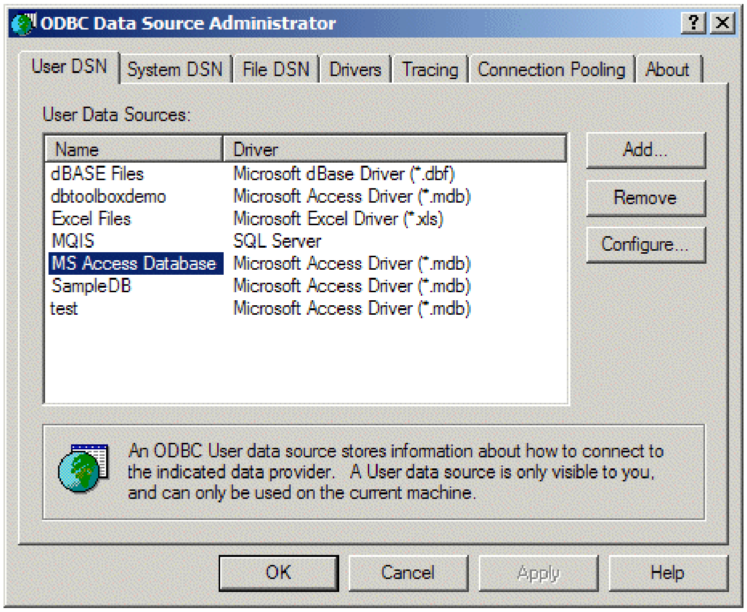
It also uses specific databases as examples. To follow along with the examples, be sure you have completed the instructions in "Prepare to Use Examples" on page 1-14.

1. Close the database in the database program. For the examples, if Microsoft Access is open, be sure to close the databases `tutorial.mdb` and `Nwind.mdb`.

2. Access the Windows Data Source Administrator dialog box in either of these ways:

   - From MATLAB, start Visual Query Builder by running `querybuilder`. Then from VQB, select **Query > Define ODBC data source**.
   - From the Windows **Start** menu, select **Settings > Control Panel > Administrative Tools > Data Sources (ODBC)**.

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



User Data Sources:

Name	Driver
dBASE Files	Microsoft dBase Driver (*.dbf)
dbtoolboxdemo	Microsoft Access Driver (*.mdb)
Excel Files	Microsoft Excel Driver (*.xls)
MQIS	SQL Server
MS Access Database	Microsoft Access Driver (*.mdb)
SampleDB	Microsoft Access Driver (*.mdb)
test	Microsoft Access Driver (*.mdb)

An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you, and can only be used on the current machine.

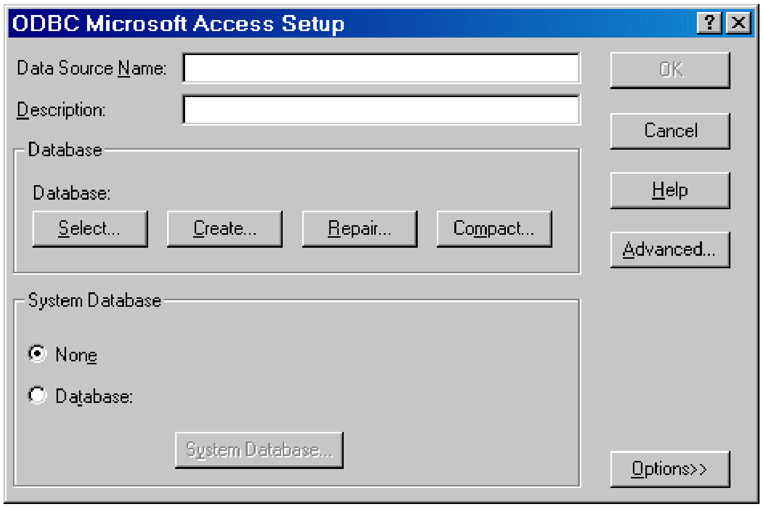
3 Select the **User DSN** tab. A list of existing user data sources appears.

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

5 Select the ODBC driver that the data source you are creating will use and click **Finish**.

- For the examples in this book, select `Microsoft Access Driver (*.mdb)`.
- Otherwise, select the driver for your database.

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



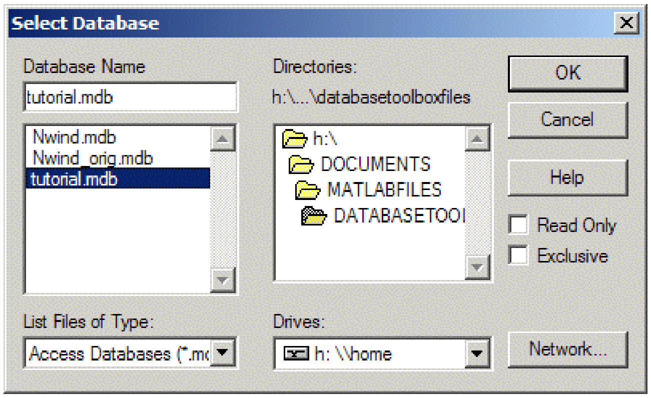
6 Provide a **Data Source Name** and **Description**.

- For the first sample data source, type `dbtoolboxdemo` as the **Data Source Name** in order to follow along with the examples in this documentation. For the **Description**, enter `tutorial database`.
- 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` as shown in the preceding illustration. You specified its location as part of "Prepare to Use Examples" on page 1-14.

If your database resides on another system to which your PC is connected, you must first click **Network** in the Select Database dialog box. The Map Network Drive dialog box appears. Find and select the directory containing the database you want to use, and then click **Finish**.

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

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

9 Repeat steps 4 through 8 to define the data source for the other example database, `Nwind`.

- In step 6, type `SampleDB` as the **Data Source Name**, and `Northwind database` as the **Description**.
- In step 7, select `Nwind.mdb`. For more information, see "SampleDB Data Source" on page 1-14.

10 Click **OK** to close the ODBC Data Source Administrator dialog box, which now contains the `dbtoolboxdemo` and `SampleDB` data sources. If VQB is open, close it to see the data sources you just added.

View All Data Sources. Use `getdatasources` to view the names of all valid ODBC and JDBC data sources.

Setting Up a Data Source for JDBC Drivers

To set up a data source using JDBC drivers, you include a reference in a MATLAB Java classpath file that specifies the location of the JDBC drivers file. To use VQB with JDBC drivers, you must then define the data source. These steps provide the instructions:

1 "Find Your JDBC Drivers Filename" on page 1-20.

2 "Include the Reference in the MATLAB Java Classpath" on page 1-21.

3 "Define a JDBC Data Source in Visual Query Builder" on page 1-23 to use Visual Query Builder with JDBC drivers.

   If you are using Database Toolbox functions, you define the data source as part of the `database` function to establish the connection.

Find Your JDBC Drivers Filename

The filename that contains the JDBC drivers is different for each database system. The file is available from your database provider. Consult your database administrator if you do not know where the file is located.

Following are some examples of filenames for a few databases. Because The MathWorks does not provide these files, this information might not be correct if the database provider has changed the filenames:

Database	Filename Containing JDBC Drivers
Microsoft SQL Server	`msbase.jar`, or `mssqlserver.jar`, or `msutil.jar`
MySQL	`mysql-connector-java-`*`n.n`*
Oracle	`classes111.zip`

For some databases, you first need to unpack the compressed file containing the JDBC drivers before you can point to it in the MATLAB Java classpath file. For example, if you add a ZIP file and cannot establish a connection, try unzipping the ZIP file and adding the unzipped file instead. You can use the MATLAB `unzip` function.

For some examples of JDBC driver names contained in a drivers file, see the `database` reference page.

Include the Reference in the MATLAB Java Classpath

After identifying the JDBC drivers filename as described in "Find Your JDBC Drivers Filename" on page 1-20, you must specify its location in the MATLAB Java classpath. The MATLAB Java classpath consists of two segments: a static segment stored in `classpath.txt`, and a dynamic segment. You can include the reference to the JDBC drivers file in either the static segment or the dynamic segment of the MATLAB Java classpath:

- Static — See "Update and Save (Static) classpath.txt" on page 1-21
- Dynamic — See "Dynamically Update the MATLAB Java Classpath" on page 1-22

Update and Save (Static) classpath.txt. Update and save the changes to the file *`matlabroot`*`/toolbox/local/classpath.txt` when you want to access a database regularly in multiple MATLAB sessions. You only have to perform this task once and MATLAB remembers the location in all future sessions. This example uses an Oracle database system that includes the JDBC drivers in the `classes111.zip` file. Substitute the full path and filename for your database system's JDBC drivers file.

1. You can directly reference the drivers file in `classpath.txt`—skip to step 2. Or, you can copy the drivers file into a directory in your *`matlabroot`* and point to that location. The *`matlabroot`* directory is where MATLAB is installed on your system, as determined by running the `matlabroot` function in the MATLAB Command Window.

   For example, create the directory `dbtools` in *`matlabroot`*`/toolbox/local`. Copy the database drivers file, for example, `classes111.zip`, into `dbtools`.

2 Add the drivers file (for example, `classes111.zip`) to the *`matlabroot`*`/toolbox/local/classpath.txt` file by including this line in `classpath.txt` that specifies the drivers file location:

   ```
   FullPathtoJDBCDriversFilename
   ```

   For example, add the following line in `classpath.txt`:

   ```
   matlabroot/toolbox/local/dbtools/classes111.zip
   ```

   Then, to point directly to a JDBC drivers file for MySQL, add this line in `classpath.txt`:

   ```
   D:/mysql/mysql-connector-java-3.0-bin.jar
   ```

3 Restart MATLAB before accessing the database.

If the drivers file (for example `classes111.zip`) is not located where `classpath.txt` indicates, MATLAB will not display errors but cannot establish a database connection. Be sure to update `classpath.txt` with the correct location and filename for your drivers file if the information changes. If MATLAB is running when you make changes to `classpath.txt`, be sure to restart MATLAB.

Dynamically Update the MATLAB Java Classpath. Dynamically update the MATLAB Java classpath when you only want to access a database in the current session or a few other sessions. The changes are not saved after you quit MATLAB, so you perform this task during each MATLAB session in which you want to access the database.

To dynamically add the JDBC drivers file to the MATLAB Java classpath, in the MATLAB Command Window, run

```
javaaddpath FullPathtoJDBCDriversFile
```

This example adds an Oracle `classes111.zip` file:

```
javaaddpath K:/databasetools/classes111.zip
```

This example adds a MySQL JAR file:

```
javaaddpath I:/mysql/mysql-connector-java-3.0/mysql-connector-java-3.0-bin.jar
```

Note that the first time you establish a connection via the JDBC drivers after using `javaaddpath`, you might notice a delay because MATLAB searches the entire static Java classpath before searching the dynamic portion.

Define a JDBC Data Source in Visual Query Builder

After pointing to the JDBC drivers filename in the MATLAB Java classpath as described in "Include the Reference in the MATLAB Java Classpath" on page 1-21, you need to define the JDBC data source to use it with Visual Query Builder. (If you use Database Toolbox functions instead of VQB to access databases via JDBC drivers, you instead define the data sources when you connect to the database as part of the `database` function.)

See also

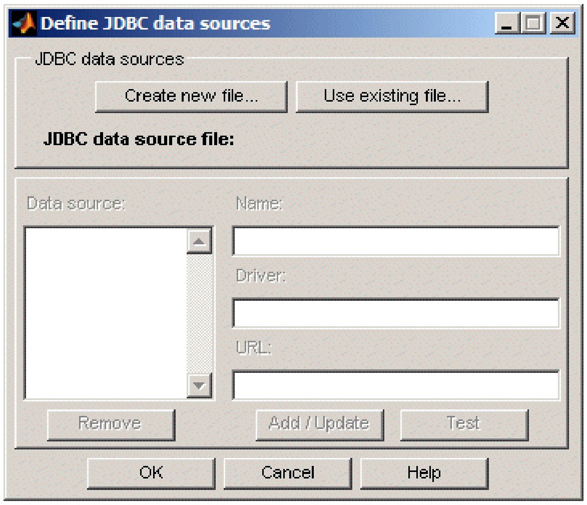
- "Using an Existing JDBC Data Source" on page 1-27
- "Function Equivalent for Using an Existing JDBC Data Source" on page 1-28
- "Making Changes to JDBC Data Sources" on page 1-28
- "Troubleshooting JDBC Drivers Problems" on page 1-29

Perform these steps to define the JDBC data source:

1. Start VQB by running `querybuilder`. Select **Query > Define JDBC Data Source**.

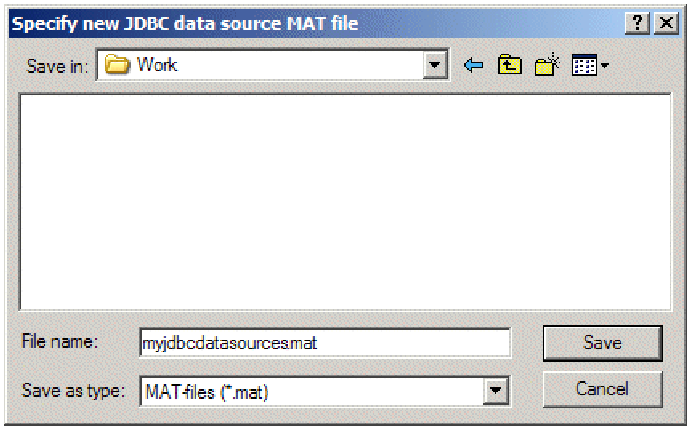
   Alternatively, you can run `confds` to open the dialog without starting VQB.

2. In the resulting Define JDBC Data Sources dialog box, click **Create New File**.



3 The Specify new JDBC data source MAT-file dialog box opens. In this dialog box, you create a MATLAB MAT-file that saves the data source information for VQB. In subsequent sessions, you recall your data source information from the file.

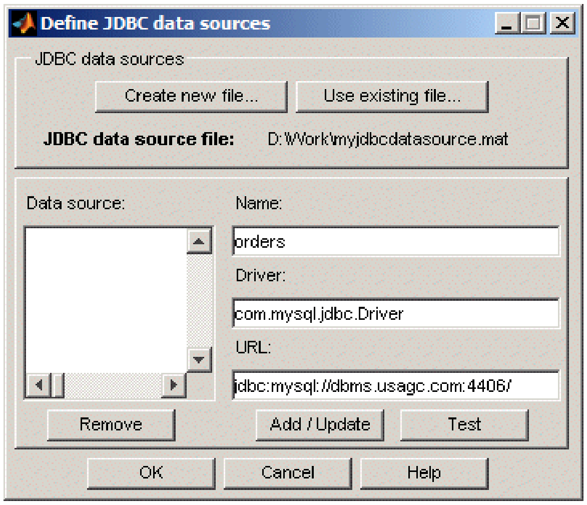
Navigate to a folder where you want to create the MAT-file, specify a name for it, including a `.mat` extension (the default value, which you must use), and click **Save**. The example shown here saves the file as `myjdbcdatasources.mat` in the `Work` directory.



4 Now in the Define JDBC Data Sources dialog box, complete the **Name**, **Driver**, and **URL** fields for your JDBC data source. Find the correct **Driver** and **URL** format in the driver manufacturer's documentation. You might need to consult with your database system administrator for the information.

- **Name**: The name you assign to the data source. For some databases, the **Name** must exactly match the name of the database as recognized by the machine it runs on.
- **Driver**: The JDBC driver name (sometimes referred to as the class that implements the Java SQL driver for your database).
- **URL**: The JDBC URL object, of the form `jdbc:`*`subprotocol`*`:`*`subname`*. The *`subprotocol`* is a database type, such as `oracle`. The *`subname`* might contain other information used by **Driver**, such as the location of the database and/or a port number. The *`subname`* might take the form *`//hostname`*`:`*`port/databasename`*.

Some sample **Driver** and **URL** strings are listed in the reference page for the `database` function under "Example 3 — Establish JDBC Connection" on page 5-30.



JDBC data source file: D:\Work\myjdbcdatasource.mat

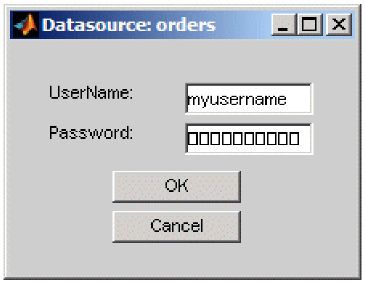
Name: orders

Driver: com.mysql.jdbc.Driver

URL: jdbc:mysql://dbms.usagc.com:4406/

5 Test the connection by clicking the **Test** button. This is optional, but recommended.

If your database requires a username and password, a dialog box appears prompting you to supply them. Enter the values in both fields and click **OK**.



- If all information is correct, a confirmation dialog box appears stating that the connection was successful. Note that if you used the `javaaddpath` method for pointing to the JDBC drivers file, you might notice a delay when testing the connection because it is the first access.
- If any of the information is incorrect, an error dialog box appears, reporting an error such as the JDBC driver was not found or loaded.

Note that if you click **Cancel** in the username dialog box, an error dialog appears. Click **OK** to close it.

6 In the Define JDBC Data Sources dialog box, click **Add/Update**. The data source now appears in the **Data source** list in the dialog box.

7 To add more data sources, repeat steps 4 through 6 for each new data source. You can add more data sources to it at a later point by editing the MAT-file.

- Be sure there is a reference to the JDBC drivers file in the MATLAB Java classpath for data sources you add, as described in “Include the Reference in the MATLAB Java Classpath” on page 1-21. For example, if you have two different MySQL data sources, you need only one reference, but if you also want to use an Oracle data source, you need a reference to its drivers file as well.
- You can create a different data source MAT-file to add new data sources. But in VQB, you can only access data sources from one MAT-file at a

time. To easily access multiple data sources from VQB, include them in a single MAT-file.

8 Click **OK** to close the Define JDBC Data Sources dialog box.

9 The data sources you just added now appear in the **Data source** list in VQB, replacing any other JDBC data sources that were listed. For instructions about using VQB, see Chapter 2, "Visual Query Builder".

10 The JDBC data sources only appear for the current MATLAB session. To access the data sources you just defined in a new MATLAB session, follow the instructions at "Using an Existing JDBC Data Source" on page 1-27.

Using an Existing JDBC Data Source. After defining a data source, you can access it in future sessions by following these steps:

1 From VQB, select **Query > Define JDBC data source**.

2 In the resulting Define JDBC Data Sources dialog box, click **Use Existing File**.

3 In the resulting Specify Existing JDBC Data Source MAT-file dialog box, navigate to the MAT-file that contains the data sources you want to use, select the MAT-file, and click **Open**.

   The data sources in the selected MAT-file appear in the Define JDBC Data Sources dialog box.

4 Click **OK** to close the Define JDBC Data Sources dialog box. The data sources now appear in the VQB **Data source** list, replacing any other JDBC data sources that were listed.

   You can only access data sources from one MAT-file at a time. To access data sources from another MAT-file, close the Define JDBC Data Sources dialog box and start again. To easily access multiple data sources from VQB, include them in a single MAT-file.

Function Equivalent for Using an Existing JDBC Data Source. After defining a data source, you can access it in future sessions using a function instead of VQB by running

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

For example, run

```
setdbprefs('JDBCDataSourceFile','D:/Work/myjdbcdatasources.mat')
```

You can include this statement in a MATLAB `startup` file to set the JDBC data source automatically when MATLAB starts.

Making Changes to JDBC Data Sources.

1 Access the existing data source. From VQB, select **Query > Define JDBC data source**.

2 In the resulting Define JDBC Data Sources dialog box, click **Use Existing File**.

3 In the resulting Specify Existing JDBC Data Source MAT-File dialog box, navigate to the MAT-file that contains the data sources you want to use, select the MAT-file, and click **Open**.

   The data sources in the selected MAT-file appear in the Define JDBC Data Sources dialog box.

4 Make changes as follows:

   - To make changes to an existing data source, select it from the list of data sources in the Define JDBC Data Sources dialog box and modify the data in the **Driver**, and **URL** fields. Click **Add/Update**.
   - To add a new data source to the MAT-file, complete the **Name**, **Driver** and **URL** fields. Click **Add/Update**.
   - To remove a data source from the MAT-file, click **Remove**. If that was the only data source in the MAT-file, delete the MAT-file too because it no longer contains useful data.

5 Click **OK** to accept the changes and close the Define JDBC Data Sources dialog box.

Troubleshooting JDBC Drivers Problems. If a data source does not appear in the VQB list, or if when you select it you receive an error dialog box or error in the MATLAB Command Window, it might be because you ran `clear all` after defining a JDBC data source where the drivers file was added using the `javaaddpath` method. In that event, redefine the data source by following the instructions at "Using an Existing JDBC Data Source" on page 1-27.

Another reason you might see an error is because the database is unavailable or there are problems with the connection. In that event, try to select the data source in VQB again, and if still unsuccessful, contact your database administrator.

If you specified an existing data source using `setdbprefs`, close VQB and reopen it so it reflects the data source changes.

Starting Database Toolbox

Use Database Toolbox functions the way you would use any MATLAB function in the Command Window. For more information, see Chapter 3, "Using Functions in Database Toolbox".

To start the Visual Query Builder GUI, type `querybuilder`. For more information about the tool, see Chapter 2, "Visual Query Builder".

Online Help

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

For a printable version of the documentation, use the PDF version on the .

Visual Query Builder

Visual Query Builder is a graphical user interface (GUI) for exchanging data between a database and MATLAB.

Getting Started with the Visual Query Builder GUI

Visual Query Builder (VQB) is an easy-to-use graphical user interface (GUI) for exchanging data with your database. With VQB, you build queries to retrieve data by selecting information from lists rather than by entering MATLAB functions. VQB retrieves the data from a database and puts it in a MATLAB cell array, structure, or numeric matrix so you can process the data using the MATLAB suite of functions. With VQB, you can display information retrieved as cell arrays in relational tables, reports, and charts. You can also use VQB to export data from MATLAB into new rows in your database. Review these key topics when you start using VQB.

- "Before You Start" on page 2-3
- "Starting Visual Query Builder" on page 2-4
- "Steps for Retrieving Data with VQB" on page 2-4
- "Steps for Exporting Data with VQB" on page 2-6
- "Help and Demos for Visual Query Builder" on page 2-8
- "Quitting Visual Query Builder" on page 2-8

You can use Database Toolbox functions instead of VQB. See "Using Visual Query Builder Versus Functions" on page 1-6 for more information.

Before You Start

Before using Visual Query Builder, set up your data source, such as the sample data sources used for the examples in this documentation: the `dbtoolboxdemo` data source (`tutorial` database) and the `sampleDB` data source (`Nwind` database), both for Microsoft Access.

Instructions for setting up these examples or any data source are in "Setting Up a Data Source" on page 1-13.

If you don't have Microsoft Access, you should still be able to follow the examples because they are not complex. If your version of Microsoft Access is different from the one used for the examples, you might get different results. If your results differ, check your version of Access, and compare the table and column names in your databases to those used in the examples.

Starting Visual Query Builder

To start the Visual Query Builder interface, type

```
querybuilder
```

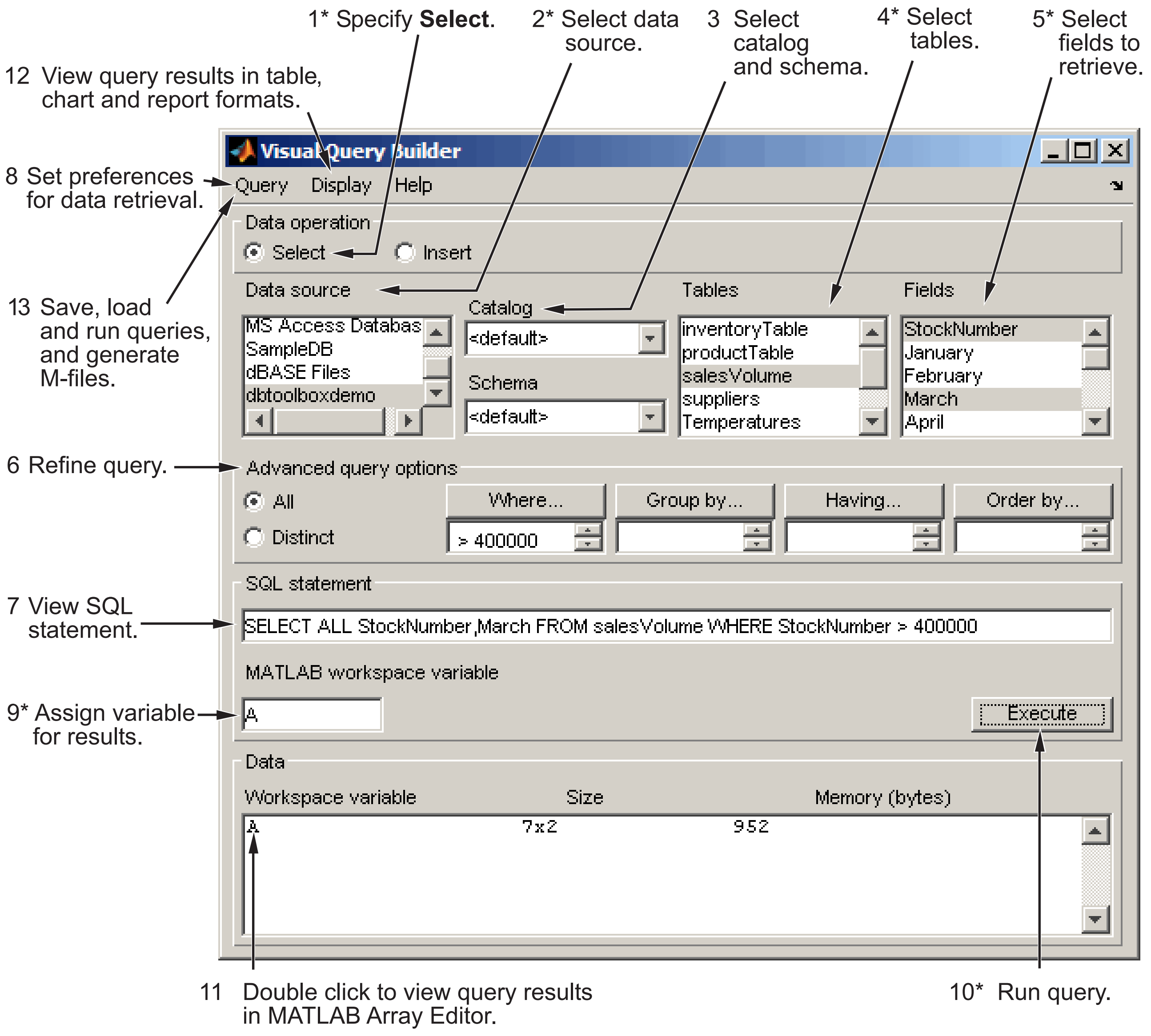
at the MATLAB prompt. Visual Query Builder opens. When you start VQB, all fields except the **Data source** are blank. The **Data source** lists the data sources you defined in “Setting Up a Data Source” on page 1-13. You can also start VQB using the **Start** menu in the MATLAB desktop.

Steps for Retrieving Data with VQB

This is a summary of the steps you take to retrieve data using VQB. Details are in subsequent topics.

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

*Required step



SELECT ALL StockNumber,March FROM salesVolume WHERE StockNumber > 400000

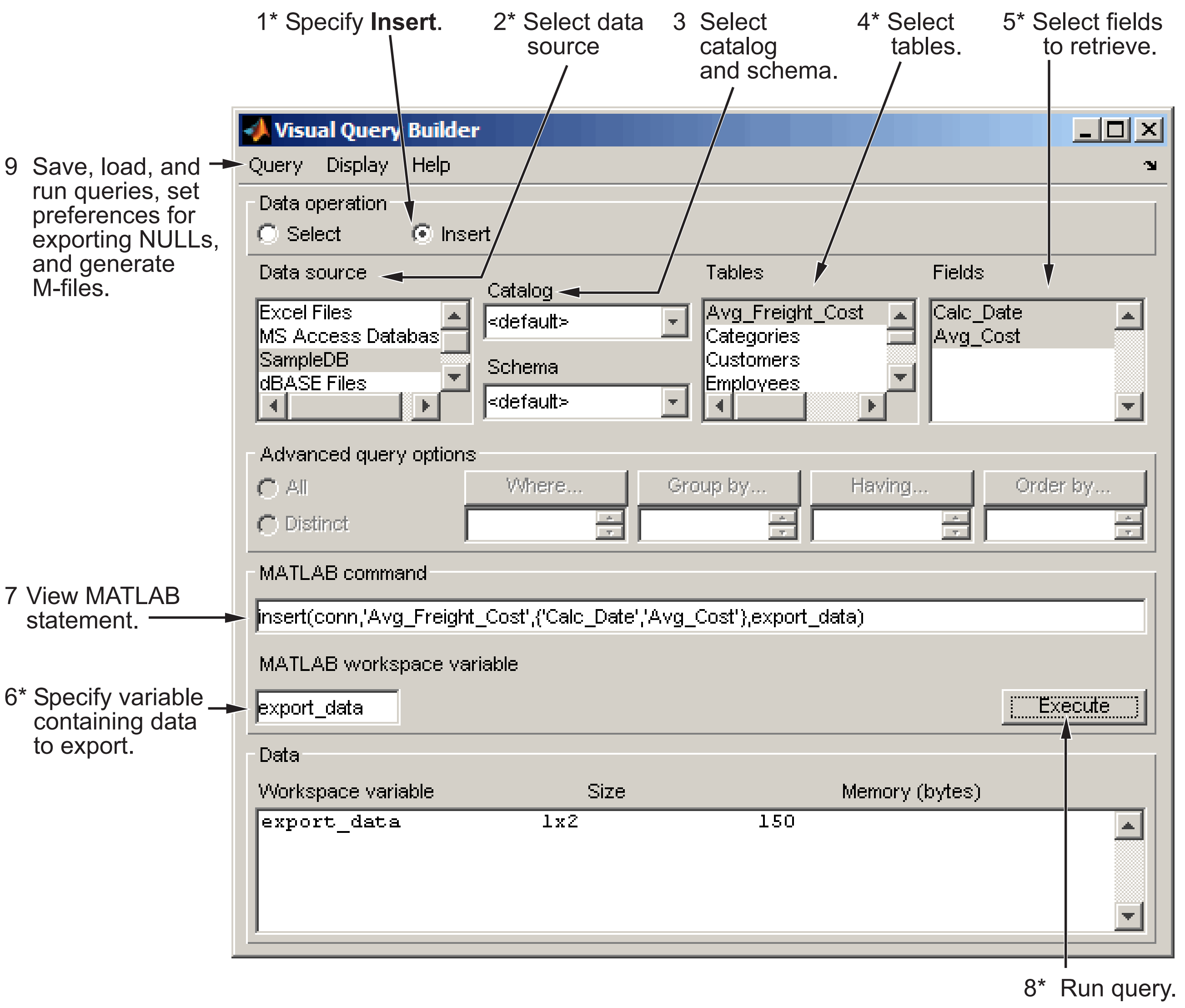
```
A                    7x2              952
```

Steps for Exporting Data with VQB

This is a summary of the steps you take to export data using VQB. Details are in “Exporting Data Using VQB” on page 2-59.

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

*Required step



1* Specify **Insert**.

2* Select data source

3 Select catalog and schema.

4* Select tables.

5* Select fields to retrieve.

6* Specify variable containing data to export.

7 View MATLAB statement.

8* Run query.

9 Save, load, and run queries, set preferences for exporting NULLs, and generate M-files.

Data operation: Select [✓ Insert]

Data source: Excel Files, MS Access Databas, SampleDB, dBASE Files

Catalog: <default>

Schema: <default>

Tables: Avg_Freight_Cost, Categories, Customers, Employees

Fields: Calc_Date, Avg_Cost

Advanced query options: All, Distinct

MATLAB command

```
insert(conn,'Avg_Freight_Cost',{'Calc_Date','Avg_Cost'},export_data)
```

MATLAB workspace variable: export_data

Data

```
Workspace variable          Size          Memory (bytes)
export_data                 1x2           150
```

Help and Demos for Visual Query Builder

Getting Help in VQB

While using Visual Query Builder, get online help by

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

For more information about getting help, see Help Browser Overview in the MATLAB documentation.

Running a Visual Query Builder Demo

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

The demo runs on Windows platforms only. It uses the `dbtoolboxdemo` data source (`tutorial` database). Instructions for setting up this data source are in "Setting Up a Data Source" on page 1-13.

Quitting Visual Query Builder

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

Creating and Running a Query to Import Data

Build and run a query to import data from your database into MATLAB, then save the query for use again later. To do this, follow these steps:

- "Before You Start" on page 2-9
- "Building and Executing a Query" on page 2-9

Before You Start

Before using VQB, set up a data source—see "Setting Up a Data Source" on page 1-13. The examples here use the `dbtoolboxdemo` data source.

Then open VQB by typing in the Command Window

```
querybuilder
```

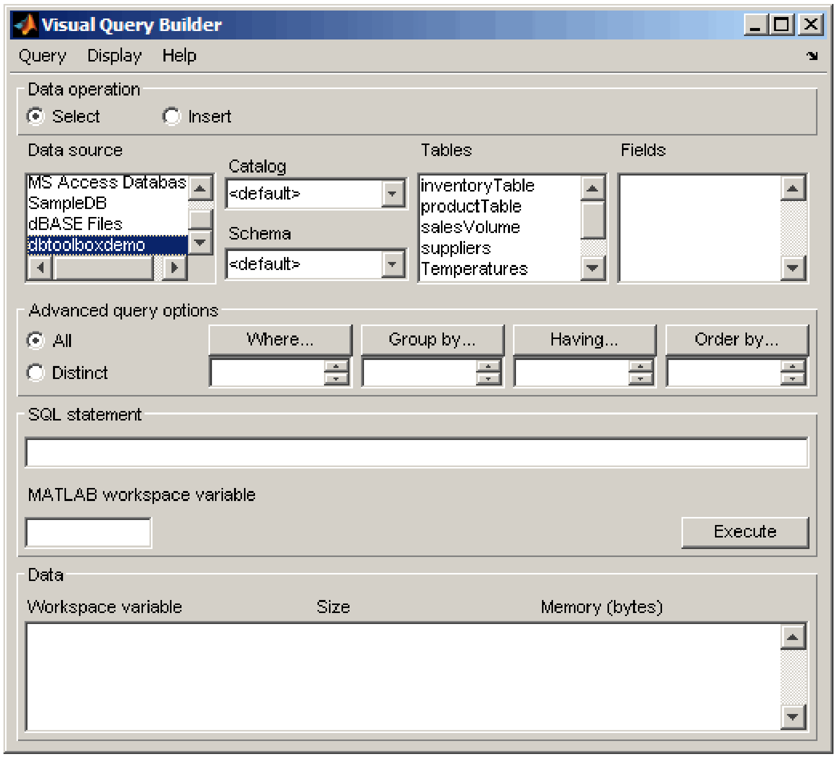
Building and Executing a Query

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

1 In the **Data operation** field, choose **Select**, meaning you want to select data from a database.

2 From the **Data source** list box, select the data source from which you want to import data. The list includes the data sources you defined in "Setting Up a Data Source" on page 1-13. Remember that JDBC data sources must be defined for each MATLAB session, and that the data sources from only a single JDBC data source MAT-file can be listed at one time.

   - For this example, select `dbtoolboxdemo`, which is the data source for the `tutorial` database.
   - If a username and password are required to access the data source, a dialog box appears prompting you to supply them. Provide the information and click **OK**. If you click **Cancel**, an error dialog box appears; click **OK** to close it. The username and password are retained only while VQB is open. If you close VQB and reopen it, you need to re-enter the username and password to access the data source.
   - After selecting a data source, the set of **Catalog**, **Schema**, and **Tables** in that data source appears.



Data operation: [✓ Select] Insert

Data source: MS Access Databas, SampleDB, dBASE Files, [✓ dbtoolboxdemo]

Catalog: <default>

Schema: <default>

Tables: inventoryTable, productTable, salesVolume, suppliers, Temperatures

Advanced query options: [✓ All] Distinct

3 Choose one of these options:

- To specify a **Catalog**, select one from the list and then select a **Schema** within that catalog. **Schema** updates to reflect your selections.
- To specify a **Schema**, select one from the list. You can select a schema after selecting a catalog. Alternatively, select a schema without selecting a catalog—in that event, the **Catalog** will be set to `default`, which is none. **Tables** updates to reflect the schema you selected.
- If you do not want to specify a **Catalog** or **Schema**, use the `<default>`, which means no catalog and schema. In effect, you skip this step.

4 From the **Tables** list box, select the table that contains the data you want to import. For this example, select `salesVolume`. Table names that include spaces appear in quotation marks. For a Microsoft Excel database, the **Tables** are Excel sheets.

After you select a table, the set of **Fields** (column names) in that table appears.

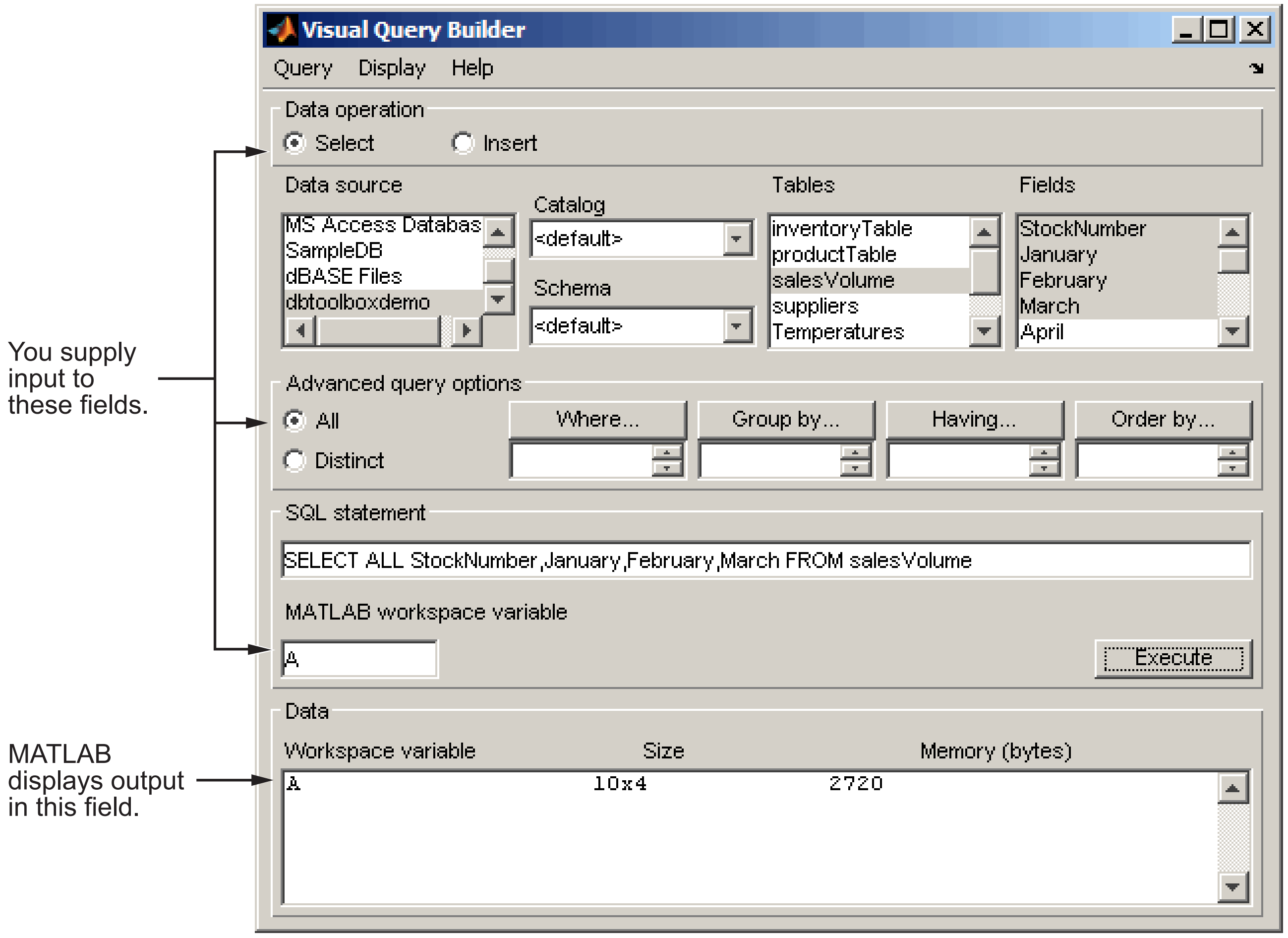
5 From the **Fields** list box, select the fields containing the data you want to import. To select more than one field, hold down the **Ctrl** key or **Shift** key while selecting. For this example, select the fields `StockNumber`, `January`, `February`, and `March`. Field names that include spaces appear in quotation marks. To deselect an entry, use **Ctrl**+click.

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

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

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

   If any of the data to be retrieved is a Java `BINARY` or `OTHER` type, for example, a bitmap image, the retrieval might be time intensive. For more information about retrieving this type of data, see "Retrieving BINARY and OTHER Java Data" on page 2-57.



Data operation: Select [✓]

Data source	Tables	Fields
MS Access Databas	inventoryTable	StockNumber
SampleDB	productTable	January
dBASE Files	salesVolume	February
dbtoolboxdemo	suppliers	March
	Temperatures	April

Catalog: <default>

Schema: <default>

Advanced query options: All [✓]

SQL statement:

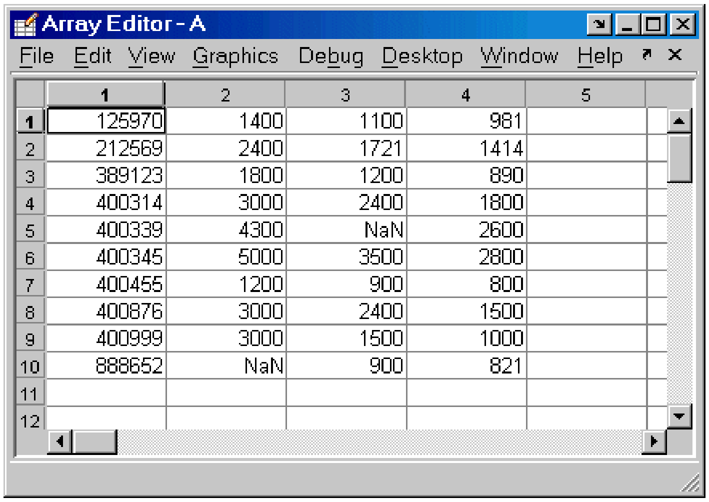
```
SELECT ALL StockNumber,January,February,March FROM salesVolume
```

MATLAB workspace variable: A

Workspace variable	Size	Memory (bytes)
A	10x4	2720

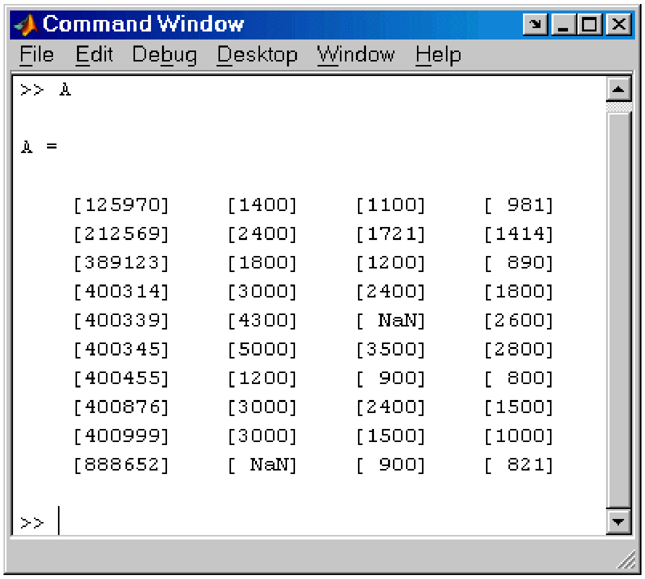
If an error dialog box appears, the query is invalid. For example, you cannot perform a query on table and field names that contain quotation marks.

8 Double-click A in the **Data** area. The contents of A are displayed in the Array Editor, where you can view and edit the data. See the MATLAB Array Editor documentation for details about using it.



	1	2	3	4	5
1	125970	1400	1100	981	
2	212569	2400	1721	1414	
3	389123	1800	1200	890	
4	400314	3000	2400	1800	
5	400339	4300	NaN	2600	
6	400345	5000	3500	2800	
7	400455	1200	900	800	
8	400876	3000	2400	1500	
9	400999	3000	1500	1000	
10	888652	NaN	900	821	
11					
12					

Another way to see the contents of A is to type A in the Command Window. For example, to read the following results, sales for item 400876 are 3000 in January, 2400 in February, and 1500 in March.



```
>> A


A =


    [125970]    [1400]    [1100]    [ 981]
    [212569]    [2400]    [1721]    [1414]
    [389123]    [1800]    [1200]    [ 890]
    [400314]    [3000]    [2400]    [1800]
    [400339]    [4300]    [ NaN]    [2600]
    [400345]    [5000]    [3500]    [2800]
    [400455]    [1200]    [ 900]    [ 800]
    [400876]    [3000]    [2400]    [1500]
    [400999]    [3000]    [1500]    [1000]
    [888652]    [ NaN]    [ 900]    [ 821]

>>
```

Note that if the data contains a Java `OTHER` data type, some fields in `A` might be empty. This happens when Java cannot pass the data through the JDBC/ODBC bridge.

Saving, Editing, and Clearing Variables for Queries

The following topics are covered in this section:

Saving a Query

After building a query in 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 in “Building and Executing a Query” on page 2-9, save the query using `basic` as the filename. Do not include spaces in the filename.

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

For a **Select** query (retrieves data), the MATLAB workspace variable name you assigned for the query results and the query preferences are *not* saved as part of the query. This protects you from inadvertently overwriting an existing variable in the MATLAB workspace when you run a saved query. For an **Insert** query (exports data), the MATLAB workspace variable name whose data you exported *is* saved as part of the query, although preferences are not saved.

Using a Saved Query

To use a saved query:

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

   The Load SQL Statement dialog box appears.

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

The VQB fields reflect the values for the saved query.

3 To run a **Select** query (imports data), assign a variable in the **MATLAB workspace variable** field and click **Execute**.

For an **Insert** query (exports data), the saved query might have included a workspace variable, which is shown as part of the **MATLAB command** field. Type that variable name or type a new name in the **MATLAB workspace variable** field. Press **Return** or **Enter** to see the updated **MATLAB command**. Click **Execute** to run the query.

See Also

You can also generate an M-file for the query that allows you to run it from the Command Window. See "Generating M-Files from VQB Queries" on page 2-68.

Editing a Query

In VQB, you can edit a query you created or opened by changing selections you made, and then executing the query again. To deselect an entry, use **Ctrl**+click.

You can also directly edit the entry in the **SQL statement** or **MATLAB command** field. After editing, you can save the query for later use.

Clearing Variables in the Data Area

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

Specifying Preferences for NULL Data, Data Format, and Error Handling

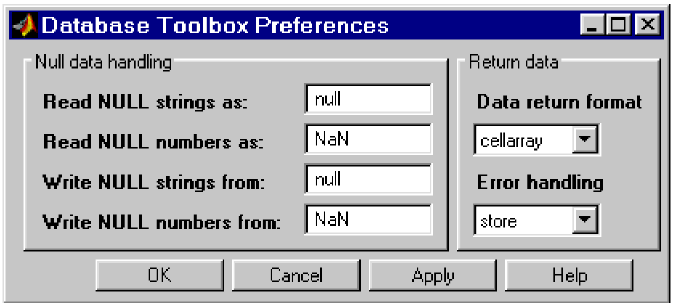
Using preferences, you can specify

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

To set preferences

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

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



Read NULL strings as: null
Read NULL numbers as: NaN
Write NULL strings from: null
Write NULL numbers from: NaN
Data return format: cellarray
Error handling: store

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

Preference	Description	New Value
Read NULL numbers as	How `NULL` numbers in a database are represented when imported into MATLAB. For the new value, `0`, the `NULL` data in the example results will appear as `0`s. Previously, they appeared as `NaN` values.	`0`
Data return format	Format for data imported into MATLAB. Select a value based on the type of data you are importing, memory considerations, and your preferred method of working with retrieved data. Cell arrays and structures support mixed data types, but require more memory and are processed more slowly than numeric matrices. Use the `numeric` format if the data you are retrieving consists only of numeric data or if the nonnumeric data is not relevant. With the `numeric` format, any nonnumeric data is converted to the representation specified in the **Read NULL numbers as** preference, for example, `NaN`. When **Read NULL numbers as** is `numeric`, the **Data return format** must also be `numeric`. For information about cell arrays, see “Working with Cell Arrays in MATLAB” on page 3-40. For information about working with strings, see “Characters and Strings” in the MATLAB Programming documentation. Because results in the example are all numeric, you can change from `cellarray` to `numeric` to reduce memory required.	`numeric`
Error handling	Behavior for handling errors when importing data. In Visual Query Builder, setting the value to `store` or `empty` means any errors are reported in a dialog box rather than in the Command Window. Set the value to `report`, which means that any errors from running the query will display immediately in the Command Window.	`report`

For more information about these preferences, see the property descriptions on the reference page for `setdbprefs`, which is the equivalent function for setting preferences. Note that there are a few other less-frequently used preferences you can specify using `setdbprefs` that are not accessible via the Preferences dialog box.

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

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

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

```
A =

      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
```

   Results are not in brackets because data is a numeric matrix rather than a cell array. `NULL` values are now represented by `0`s instead of NaNs.

Saving Preferences

Preferences apply to the current MATLAB session. They are not saved with a query. The default preferences apply when you start a new session, or after clearing all variables (for example, `clear all`). It is a good practice to verify the preference settings before you run a query.

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

Viewing Query Results

After running a query in Visual Query Builder, you can view the retrieved data by

- Typing the variable name in the MATLAB Command Window to view it there, or
- Double-clicking the variable in the VQB **Data** area to view the data in the Array Editor.

The VQB **Display** menu provides additional options for viewing data:

- “Relational Display of Data” on page 2-21
- “Chart Display of Results” on page 2-25
- “Report Display of Results in a Table” on page 2-28
- “Customized Display of Results in Report Generator” on page 2-29

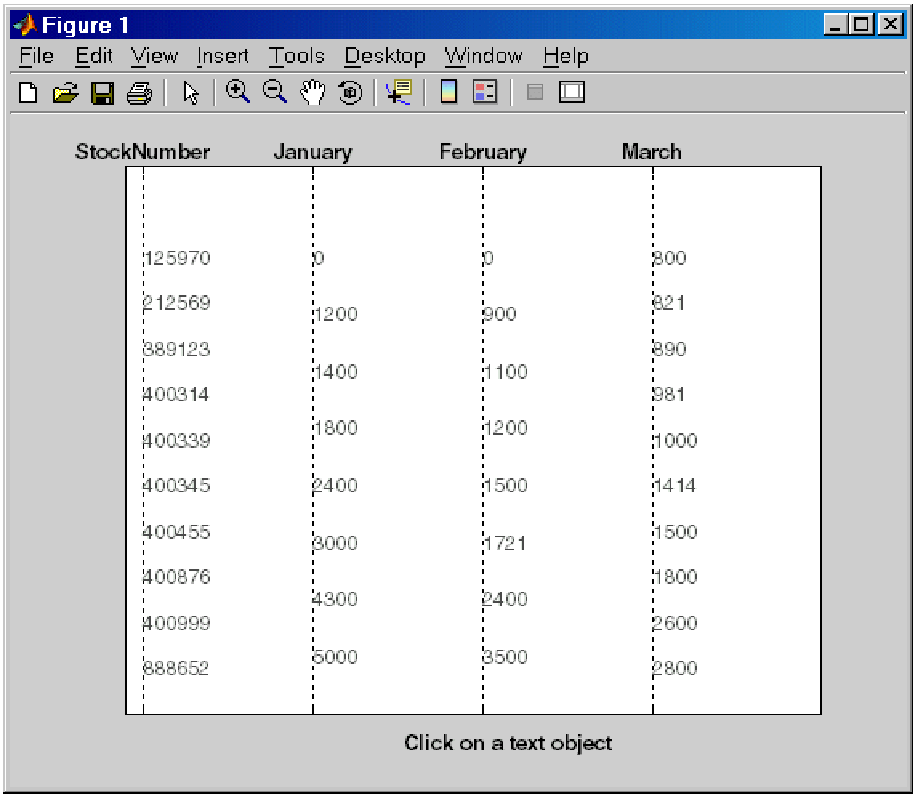
The examples in this section use the saved query from the earlier example, `basic.qry`. Use the steps below to access this query.

1. Select **Query > Preferences** and set **Read NULL numbers as** to `0`.
2. Select **Query > Load**.
3. In the Load SQL Statement dialog box, select the **File name**, `basic.qry`, and click **Open**.
4. In VQB, type a value for the **MATLAB workspace variable**, for example, `A`, and then click **Execute**.

Relational Display of Data

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

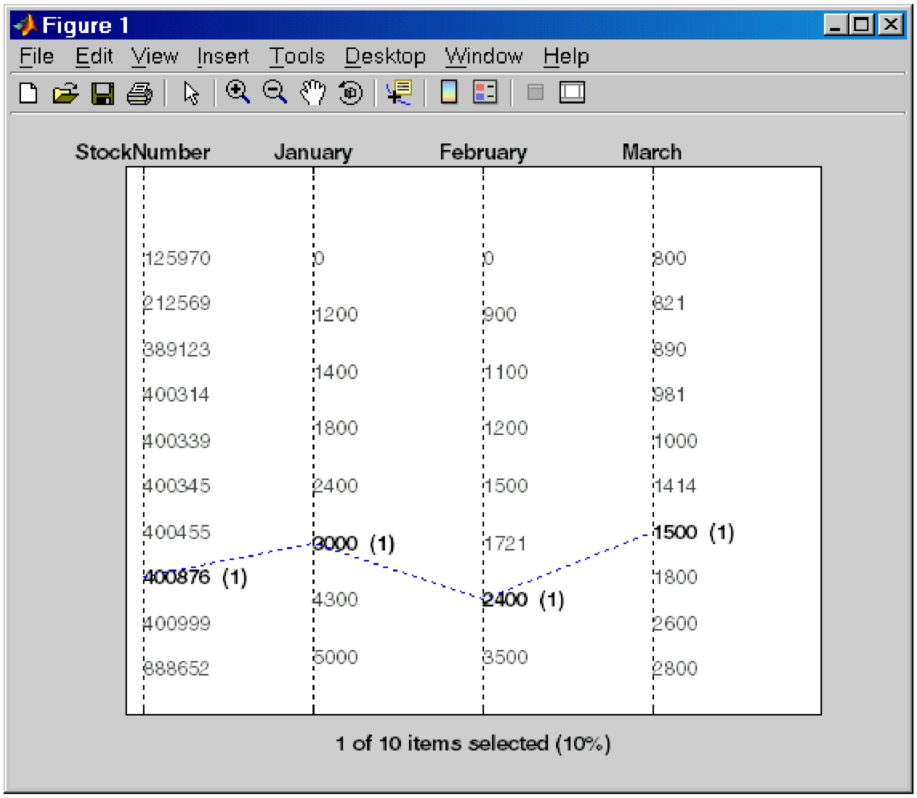
   The query results appear in a figure window.



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

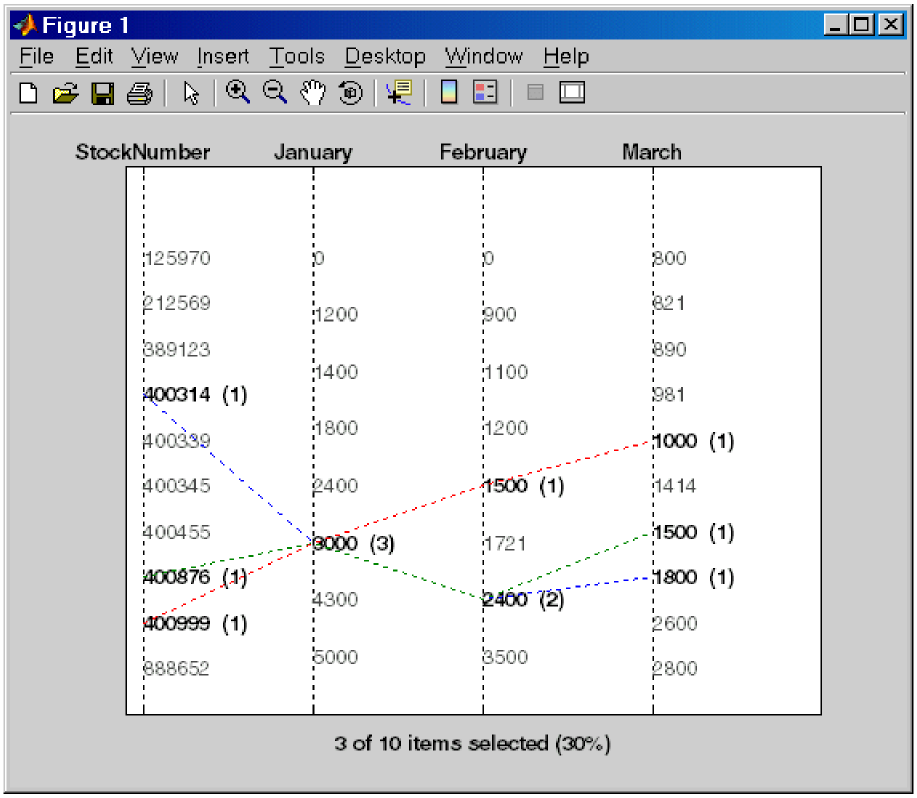
2 Click a value in the display, for example, **StockNumber** `400876`, to see the associated values.

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



1 of 10 items selected (10%)

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



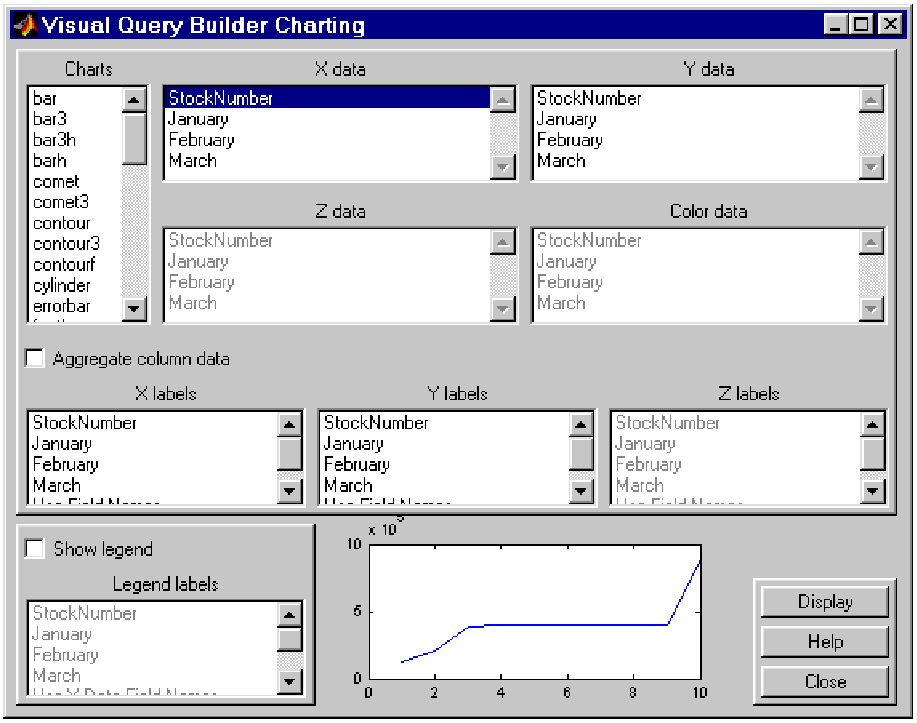
3 Because the display is presented in a MATLAB figure window, you can use some MATLAB figure features. For example, you can print the figure and annotate it. To print it, select **File > Print**. You can use other print features, such as **File > Print Preview**. For more information, use the figure window **Help** menu.

4 If the query results include many entries, the display might not effectively show all of them. You can stretch the window to make it larger, modify the query so there are fewer results, or display the results in a table (see "Report Display of Results in a Table" on page 2-28).

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** list box (`plot` is the default). For example, select `pie` to display a pie chart.

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

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

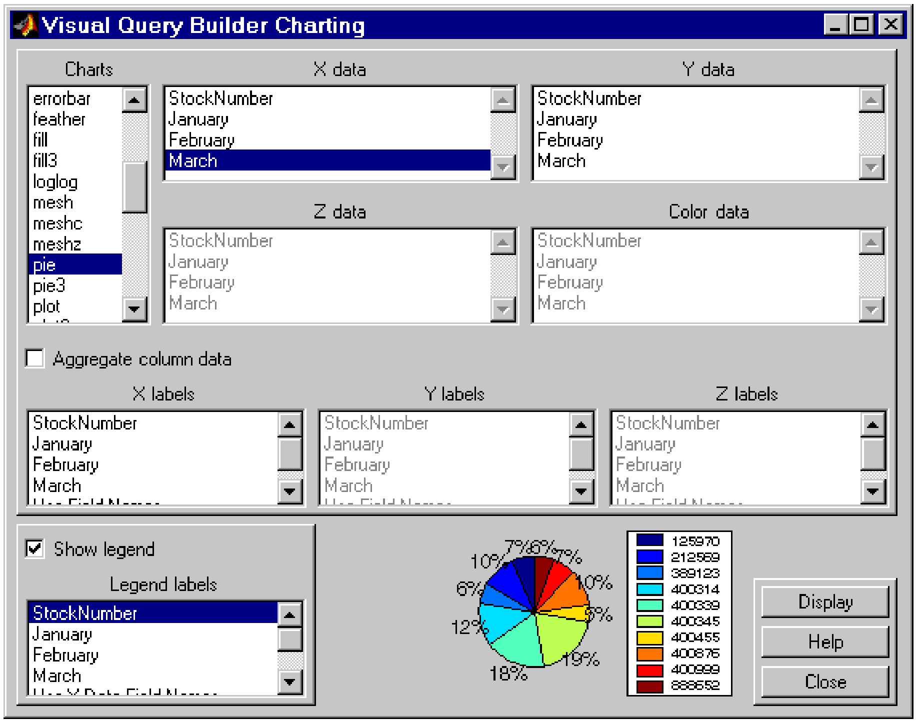
The preview of the chart at the bottom of the dialog box reflects the selection you made. For this example, the pie chart shows percentages for March data.

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

The **Legend labels** become available.

5 Select `StockNumber` from the **Legend labels** list box.

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

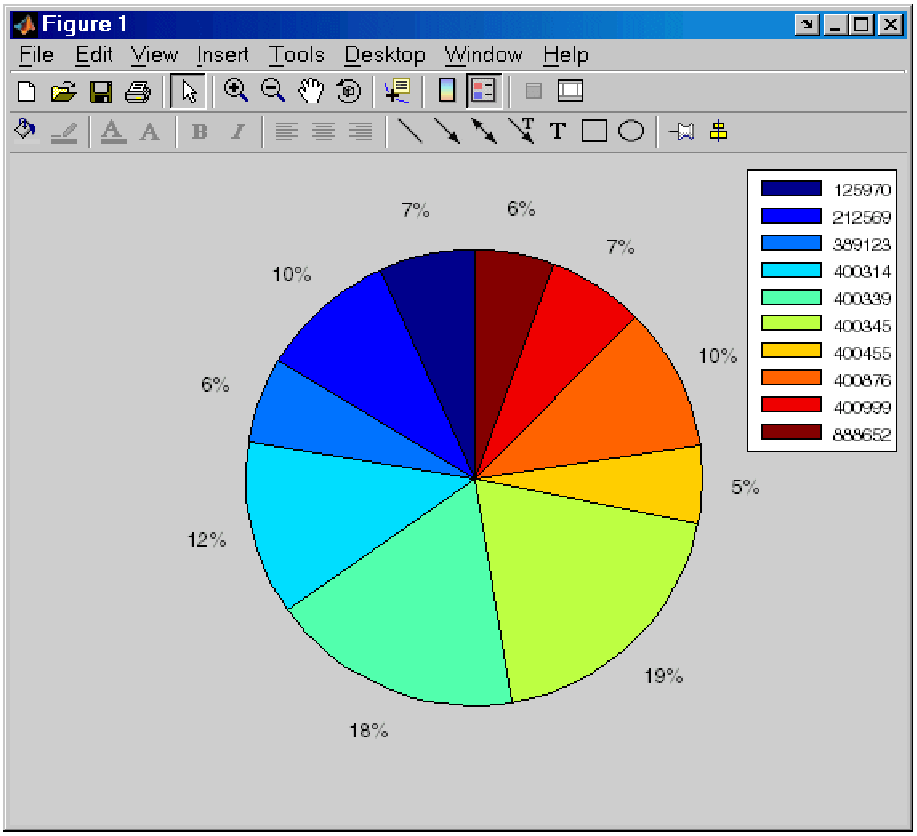


6 Click **Display**.

The pie chart appears in a figure window. Because the display is presented in a MATLAB figure window, you can use some MATLAB figure features such as printing or annotating the figure. To print the figure, select **File > Print**. You can also use **File > Print Preview**.

For example:

- Resize the window by dragging any corner or edge.
- Drag the legend to another position.
- Annotate the chart using the **Insert** menu and the annotation buttons in the Plot Edit 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, you might need to start your system Web browser before using this feature.

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

The query results appear as a table in your system Web browser.

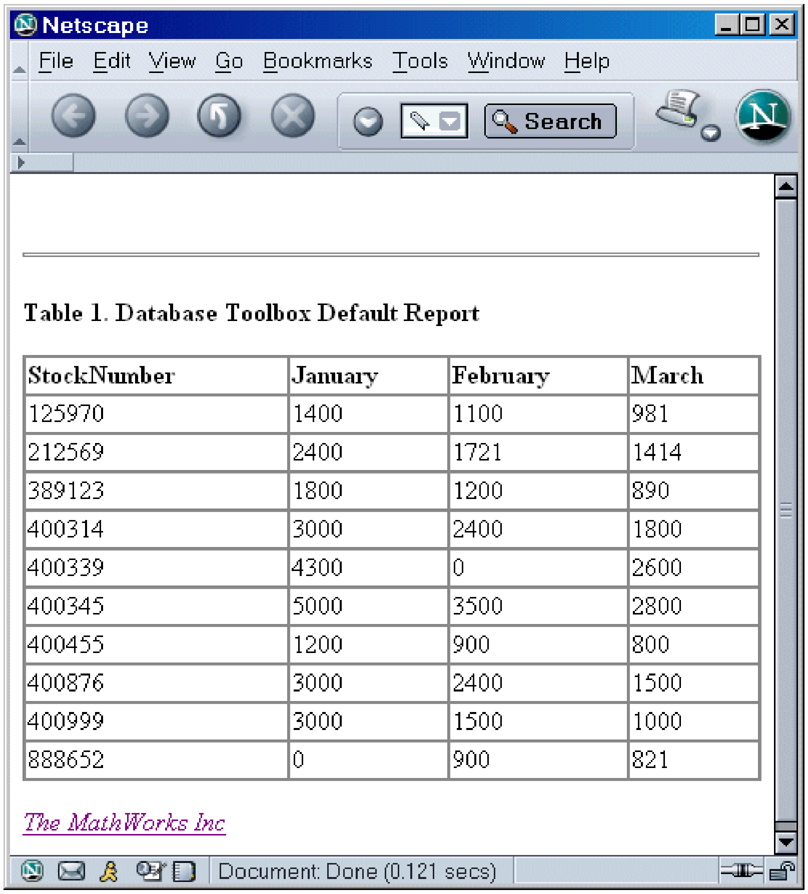


Table 1. Database Toolbox Default Report

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

The MathWorks Inc

Document: Done (0.121 secs)

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.

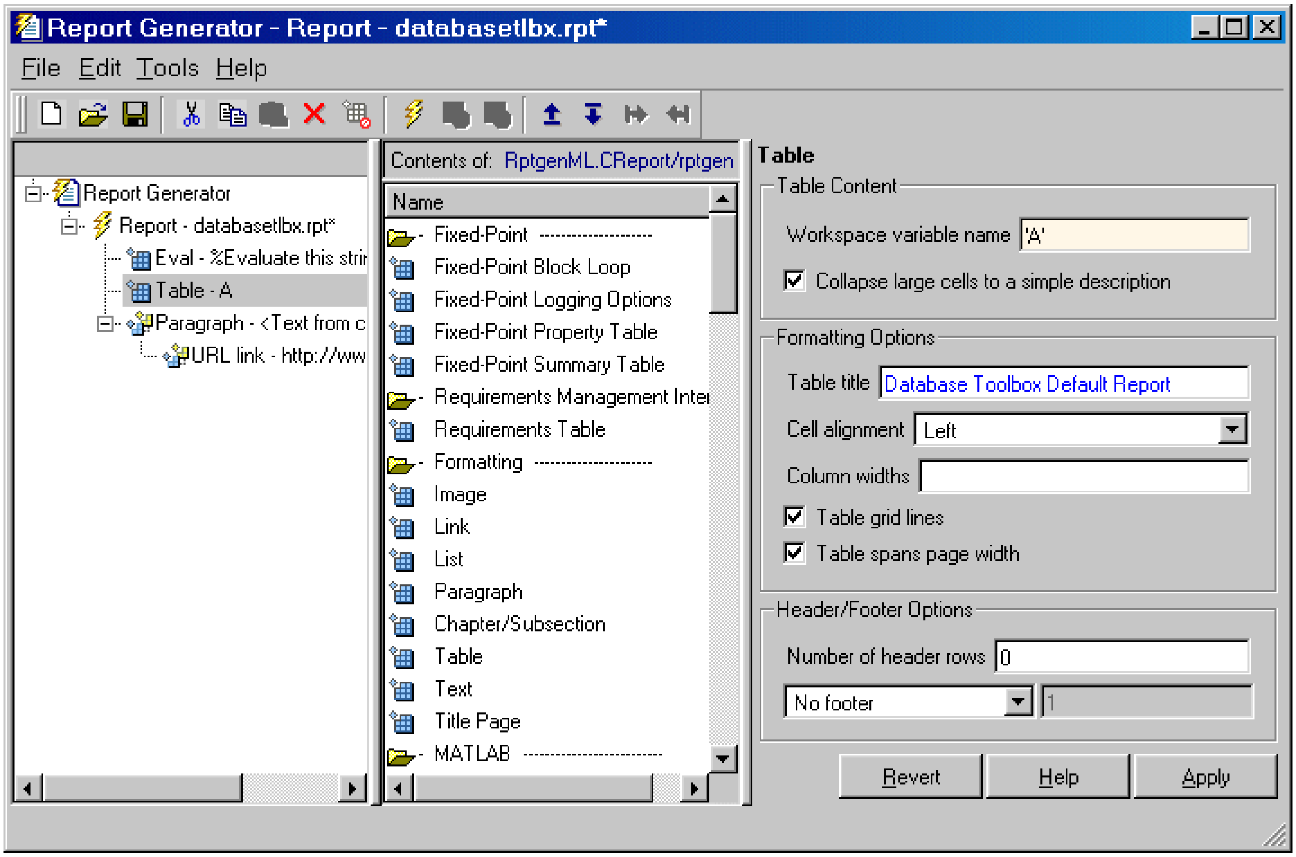
3 Use your Web browser to save the report as an HTML page if you want to view it later. To print the report, use the print features in your Web browser.

Customized Display of Results in Report Generator

If the MATLAB Report Generator is installed, you can customize the display of results using that product.

1. Because some browser configurations do not launch automatically, you might need to start your system Web browser before using this feature.
2. After executing a query, select **Report Generator** from the **Display** menu.

   The Report Generator interface opens.
3. In the contents listing, select `databasetlbx.rpt` (in *`matlabroot`*`/toolbox/database/vqb`). This is a sample report template. You can create and use your own reports.
4. Modify the report format—click **Open Report**.
   a. In the left column, under **Report Generator > Report databasetlbx.rpt**, select `Table - ans`.
   b. In the right column, under **Table Content**, for **Workspace Variable Name**, replace the default, `'ans'` with the **Workspace variable name** you had assigned to the query result in Visual Query Builder, for example, '`A`'.
   c. In the right column, under **Header/Footer Options**, set the **Number of header rows** to `0`.
   d. Click **Apply**.


Report Generator - Report - databasetlbx.rpt*
File Edit Tools Help
Report Generator
Report - databasetlbx.rpt*
Eval - %Evaluate this strir
Table - A
Paragraph - <Text from c
URL link - http://ww
Contents of: RptgenML.CReport/rptgen
Name
Fixed-Point
Fixed-Point Block Loop
Fixed-Point Logging Options
Fixed-Point Property Table
Fixed-Point Summary Table
Requirements Management Inte
Requirements Table
Formatting
Image
Link
List
Paragraph
Chapter/Subsection
Table
Text
Title Page
MATLAB
Revert
Help
Apply

Table

Table Content

Workspace variable name: 'A'

[✓] Collapse large cells to a simple description

Formatting Options

Table title: Database Toolbox Default Report

Cell alignment: Left

Column widths:

[✓] Table grid lines

[✓] Table spans page width

Header/Footer Options

Number of header rows: 0

No footer 1

Click the **Help** button in the dialog box for more information about this and other features of Report Generator.

5 To run and view the report, select **File > Generate Report**.

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

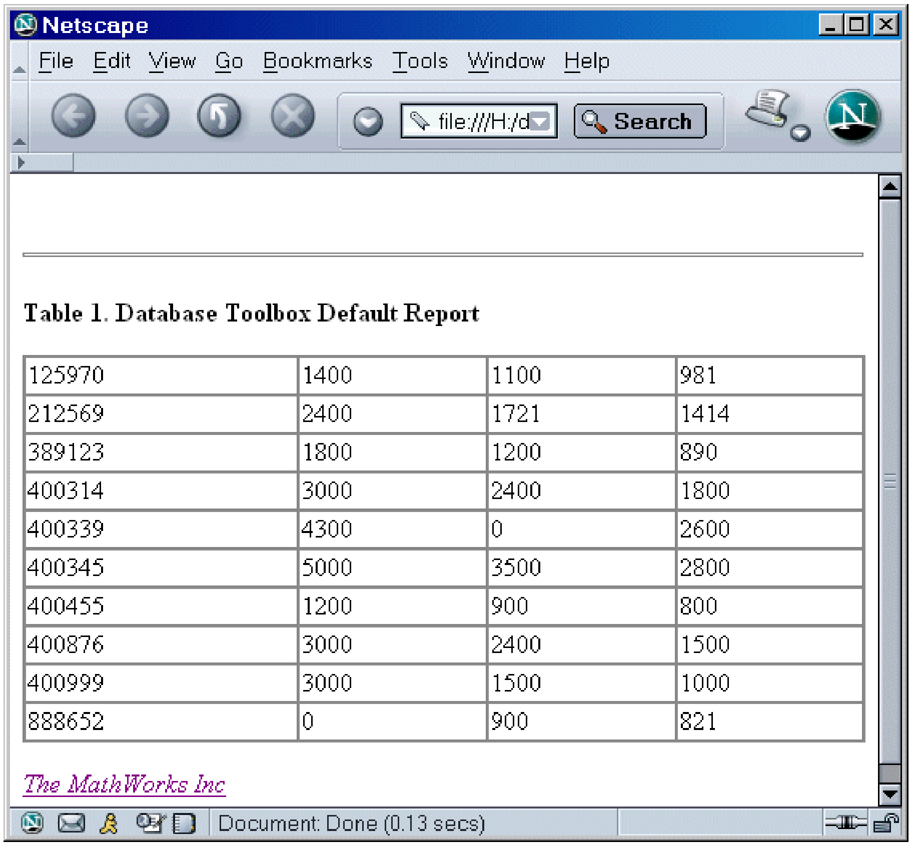


Table 1. Database Toolbox Default Report

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

The MathWorks Inc

Document: Done (0.13 secs)

6 The names of the fields from Visual Query Builder do not automatically appear as column headers in the report, as they did for the feature described in “Report Display of Results in a Table” on page 2-28. You can modify the workspace variable, for example, A, to include the field names so that they appear in the report. For example, in the Command Window, redefine A using

```
A = [{'Stock Number', 'January', 'February', 'March'};A]
```

In Report Generator, change the **Header/Footer Options**, **Number of header rows** to 1 (refer back to step 4-c for details). The output report now shows the field names as headings.

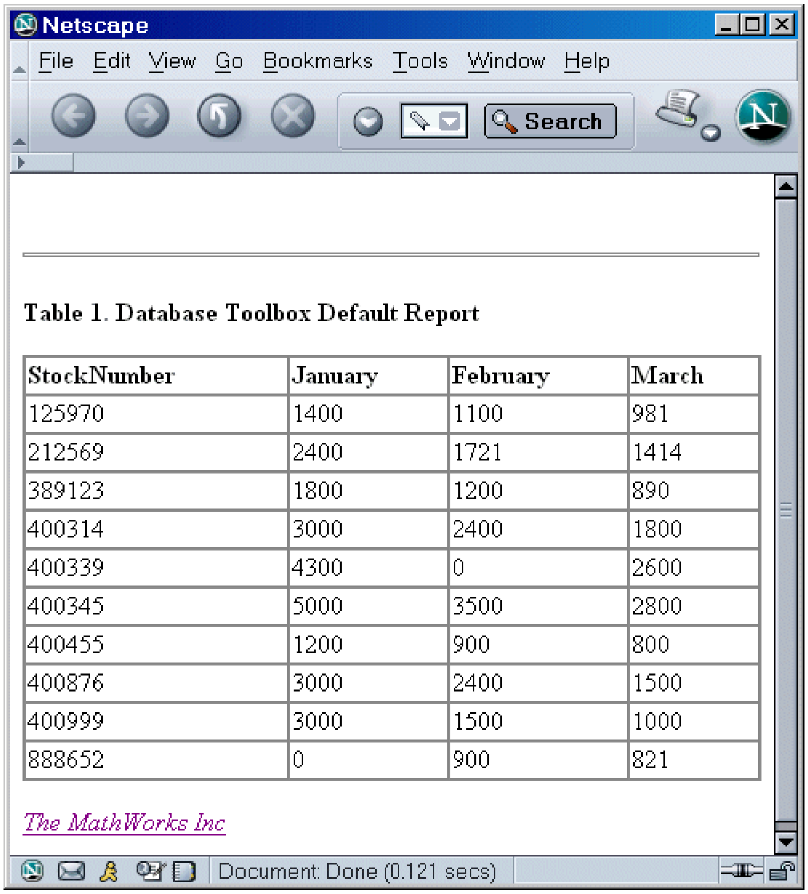

Netscape
File Edit View Go Bookmarks Tools Window Help
Search
Document: Done (0.121 secs)

Table 1. Database Toolbox Default Report

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

The MathWorks Inc

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.

Fine-Tuning Queries Using Advanced Query Options

Use advanced query options when retrieving data with Visual Query Builder for these tasks:

- “Retrieving Unique Occurrences” on page 2-34
- “Retrieving Information That Meets Specified Criteria” on page 2-36
- “Evaluating Multiple Constraints” on page 2-39
- “Presenting Results in Specified Order” on page 2-44
- “Creating Subqueries for Values from Multiple Tables” on page 2-47
- “Creating Queries for Results from Multiple Tables” on page 2-52
- “Other Features in Advanced Query Options” on page 2-56

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

Retrieving Unique Occurrences

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

1. Set preferences; for this example, set **Data return format** to `cellarray` and **Read NULL numbers as** to `NaN`.
2. For the **Data operation**, choose **Select**.
3. From **Data source**, select a data source; for this example, `dbtoolboxdemo`.
4. Do not specify **Catalog** and **Schema**. From **Tables**, select a table; for this example, `SalesVolume`.
5. From **Fields**, select the fields; for this example, `January`.

6 Run the query to retrieve all occurrences:

  a In **Advanced query options**, select **All**.

  b Assign a **MATLAB workspace variable**; for this example, `All`.

  c Click **Execute**.

7 Run the query to retrieve only unique occurrences:

  a In **Advanced query options**, select **Distinct**.

  b Assign a **MATLAB workspace variable**; for this example, `Distinct`.

  c Click **Execute**.

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

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

```
All =

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


Distinct =

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

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

Retrieving Information That Meets Specified Criteria

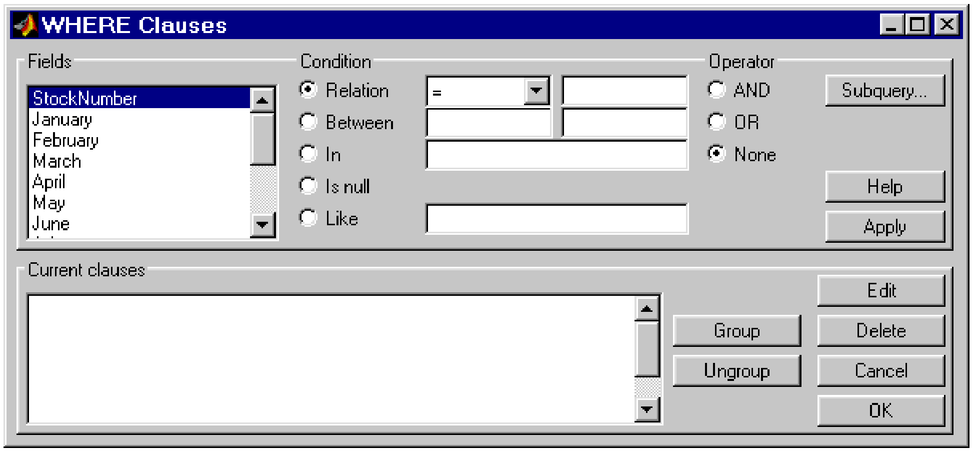
Use the **Where** field in **Advanced query options** to retrieve only the information that meets the criteria you specify. This example uses the `basic.qry` query that was created and saved as explained in “Creating and Running a Query to Import Data” on page 2-9. It limits the results to those stock numbers greater than 400000 and less than 500000:

1 Load `basic.qry`. For instructions, see “Using a Saved Query” on page 2-15.

2 Set preferences; for this example, set **Data return format** to `cellarray` and **Read NULL numbers as** to `NaN`.

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

The WHERE Clauses dialog box appears.



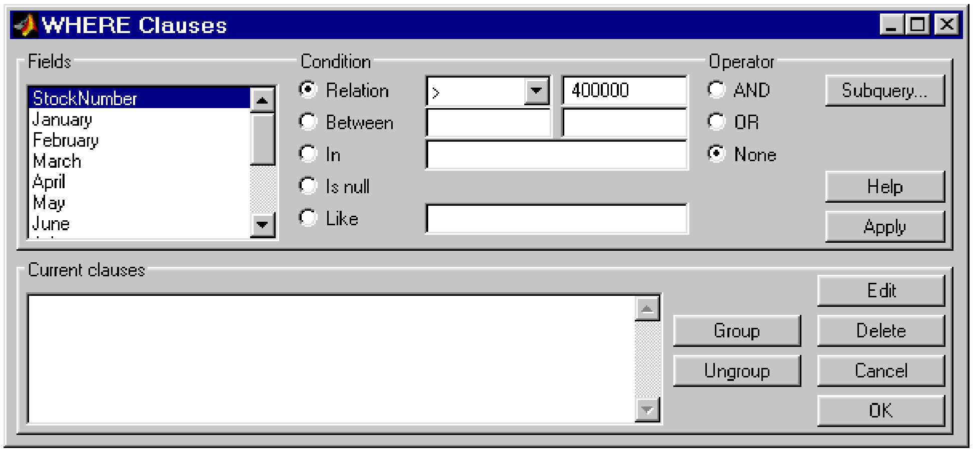
4 From **Fields**, select the fields whose values you want to restrict. For example, select `StockNumber`.

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

a Select **Relation**.

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

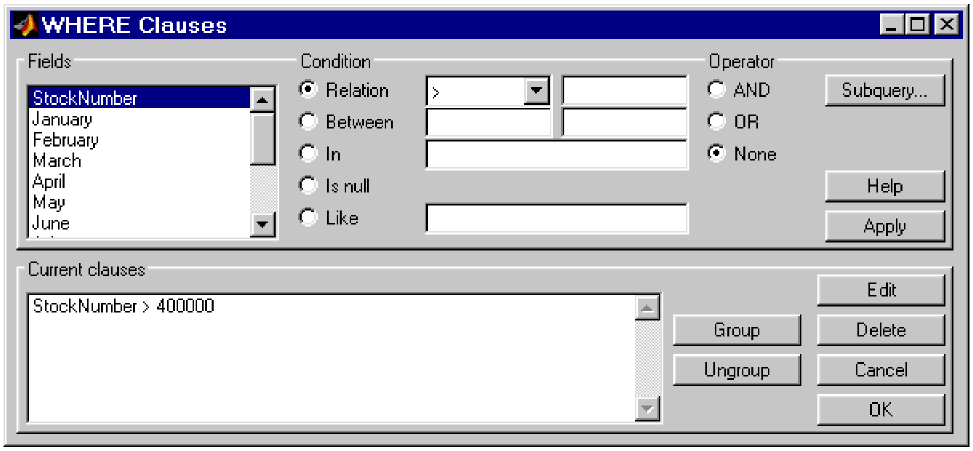
c In the field to the right of the drop-down list, type `400000`.



Condition: [✓ Relation] > 400000
Operator: [✓ None]

d Click **Apply**.

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



Condition: [✓ Relation] >
Operator: [✓ None]

Current clauses: StockNumber > 400000

6 You can add another condition. First you edit the current clause to add the `AND` operator to it, and then you provide the new condition.

   a Select `StockNumber > 400000` from **Current clauses**.

   b Click **Edit** (or double-click the`StockNumber` entry in **Current clauses**).

      The **Condition** reflects the `StockNumber` clause.

   c For **Operator**, select **AND**.

   d Click **Apply**.

      The **Current clauses** updates to show

      ```
      StockNumber > 400000 AND
      ```

7 Add the new condition. For example, specify that `StockNumber` must also be less than `500000`:

   a From **Fields**, select `StockNumber`.

   b Select **Relation** from **Condition**.

   c From the drop-down list to the right of **Relation**, select `<`.

   d In the field to the right of the drop-down list, type `500000`.

   e Click **Apply**.

      The **Current clauses** area now shows

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

8 Click **OK**.

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

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

10 Click **Execute**.

   The results are a 6-by-4 matrix.

11 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 7 in "Building and Executing a Query" on page 2-9).

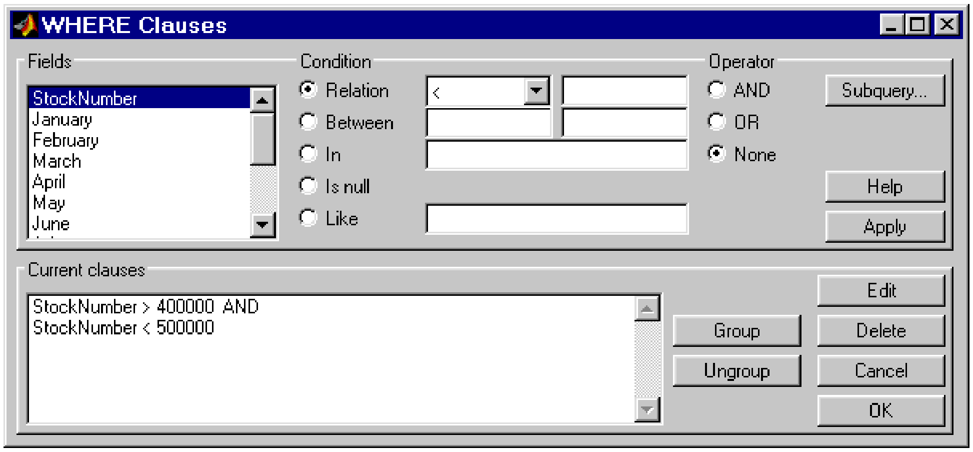
```
A =

    [400314]    [3000]    [2400]    [1800]
    [400339]    [4300]    [ NaN]    [2600]
    [400345]    [5000]    [3500]    [2800]
    [400455]    [1200]    [ 900]    [ 800]
    [400876]    [3000]    [2400]    [1500]
    [400999]    [3000]    [1500]    [1000]
```

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

Evaluating Multiple Constraints

In the WHERE Clauses dialog box, you can group together constraints so that the group of constraints is evaluated as a whole in the query. For the example, basic_where.qry, where StockNumber is greater than 400000 and less than 50000, modify the query to group constraints. The new query will retrieve results where sales in any of the 3 months is greater than 1500 units, as long as sales for each of the 3 months is greater than 1000 units.

Click **Where** in Visual Query Builder. The WHERE Clauses dialog box appears as follows, to retrieve data where the StockNumber is greater than 400000 and less than 50000.



Current clauses:

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

1 Add the criteria to retrieve data where sales in any of the 3 months is greater than 1500 units.

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

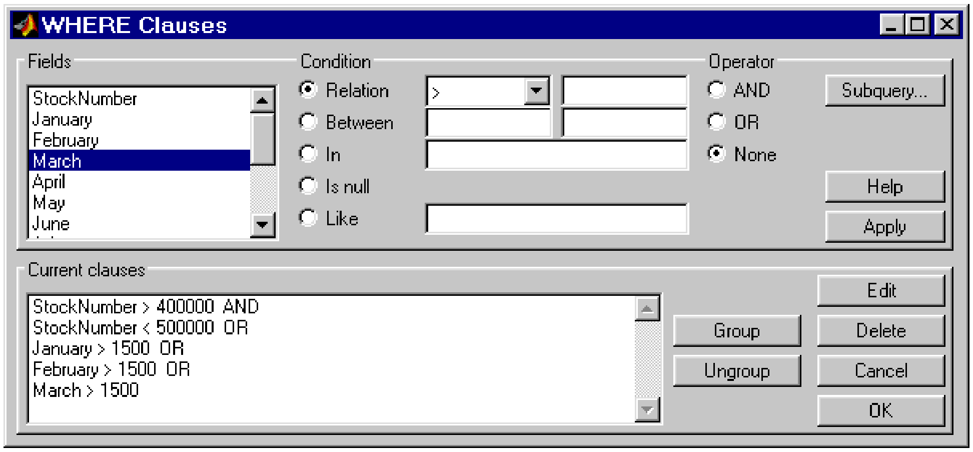
   b For **Operator**, select `OR`, and then click **Apply**.

   c In **Fields**, select `January`. For **Relation**, select `>` and type `1500` in the field for it. For **Operator**, select `OR`, and then click **Apply**.

   d In **Fields**, select `February`. For **Relation**, select `>` and type `1500` in the field for it. For **Operator**, select `OR`, and then click **Apply**.

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

   The WHERE Clauses dialog box appears as follows.



Fields: StockNumber, January, February, March (selected), April, May, June

Condition: [✓ Relation] >

Operator: [✓ None]

Current clauses:

```
StockNumber > 400000  AND
StockNumber < 500000  OR
January > 1500  OR
February > 1500  OR
March > 1500
```

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

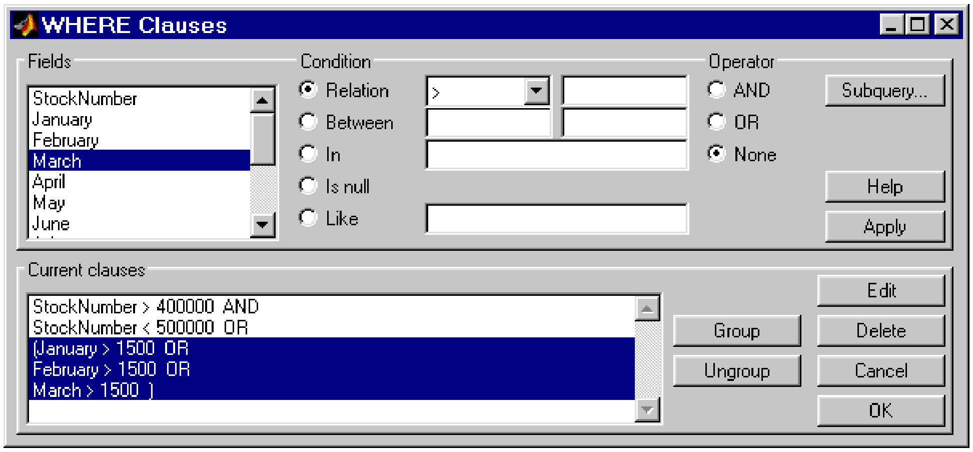
a In **Current clauses**, select the statement `January >1500 OR`.

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.



```
StockNumber > 400000 AND
StockNumber < 500000 OR
(January > 1500 OR
February > 1500 OR
March > 1500 )
```

3 Add the criteria to retrieve data where sales in each of the 3 months is greater than 1000 units:

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

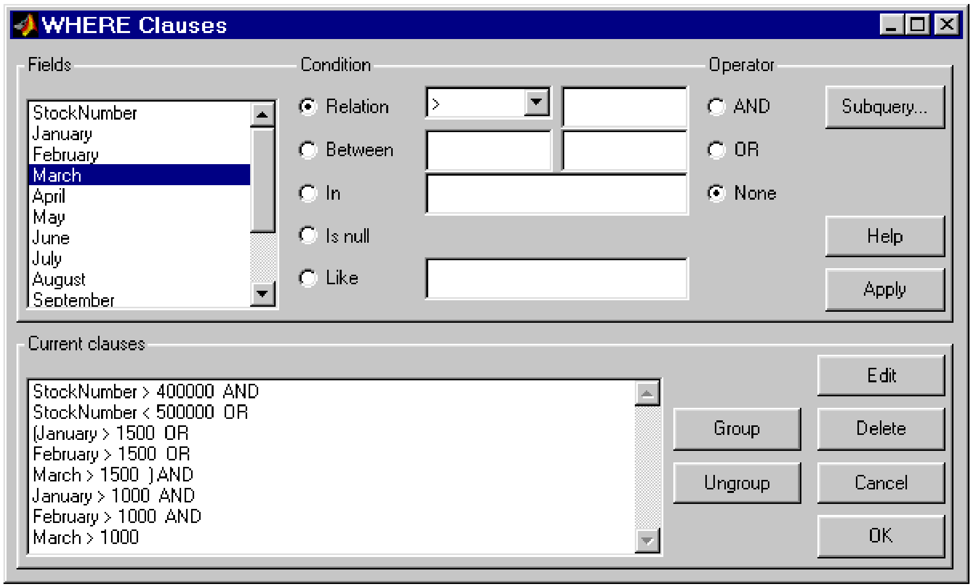
b For **Operator**, select `AND`, and then click **Apply**.

c In **Fields**, select `January`. For **Relation**, select `>` and type `1000` in the field for it. For **Operator**, select `AND`, and then click **Apply**.

d In **Fields**, select `February`. For **Relation**, select `>` and type `1000` in the field for it. For **Operator**, select `AND`, and then click **Apply**.

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

The WHERE Clauses dialog box appears as follows.



```
StockNumber > 400000  AND
StockNumber < 500000  OR
(January > 1500  OR
February > 1500  OR
March > 1500  ) AND
January > 1000  AND
February > 1000  AND
March > 1000
```

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 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. MATLAB returns

```
AA =

    [212569]    [2400]    [1721]    [1414]
    [400314]    [3000]    [2400]    [1800]
    [400339]    [4300]    [ NaN]    [2600]
    [400345]    [5000]    [3500]    [2800]
    [400455]    [1200]    [ 900]    [ 800]
    [400876]    [3000]    [2400]    [1500]
    [400999]    [3000]    [1500]    [1000]
```

Removing Grouping

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

For the above example, to remove the grouping, select

```
(January > 1000 AND
```

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

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

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

Presenting Results in Specified Order

By default, the order of the rows in the query results depends on their order in the database, which is effectively random. Use **Order by** in **Advanced query options** to specify the order in which results appear. This example uses the `basic_where.qry` query, which was created and saved in the example presented in "Retrieving Information That Meets Specified Criteria" on page 2-36.

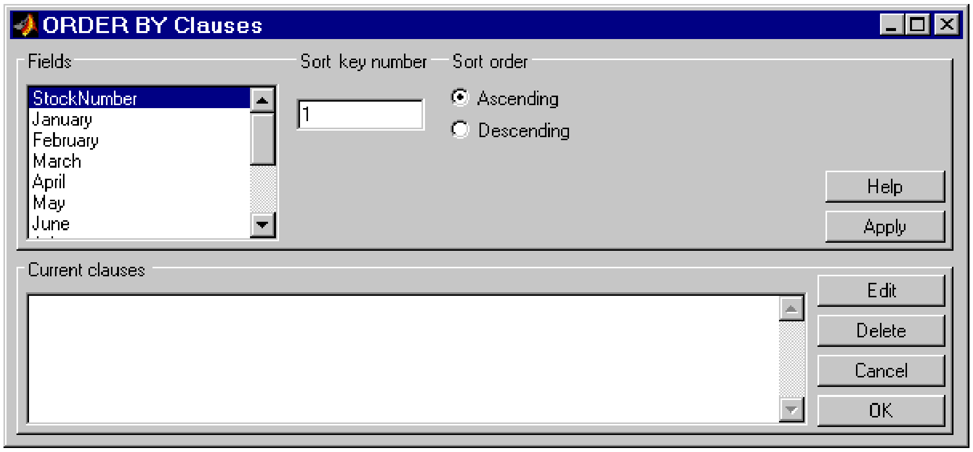
This example sorts the results of `basic_where.qry`, so that January is the primary sort field, February the secondary, and March the last. Results for January and February are ascending, and results for March are descending:

1 Load `basic_where.qry`. For instructions, see "Using a Saved Query" on page 2-15.

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

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

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



4 For the **Fields** whose results you want to specify the order of, specify the **Sort key number** and **Sort order**. For example, specify January as the primary sort field, with results displayed in ascending order:

a From **Fields**, select `January`.

b For **Sort key number**, type `1`.

c For **Sort order**, select **Ascending**.

d Click **Apply**.

The **Current clauses** area now shows

```
January ASC
```

5 Specify February as the second sort field, with results displayed in ascending order.

   a From **Fields**, select `February`.

   b For **Sort key number**, type `2`.

   c For **Sort order**, select **Ascending**.

   d Click **Apply**.

   The **Current clause**s area now shows

```
January ASC
February ASC
```

6 Specify March as the third sort field, with results displayed in descending order.

   a From **Fields**, select `March`.

   b For **Sort key number**, type `3`.

   c For **Sort order**, select **Descending**.

   d Click **Apply**.

   The **Current clause**s area now shows

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

7 Click **OK**.

The Order By Clauses dialog box closes. The **Order by** field and the **SQL statement** in Visual Query Builder reflect the order by clause you specified.

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

9 Click **Execute**.

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

```
A =

    [400314]    [3000]    [2400]    [1800]
    [400339]    [4300]    [ NaN]    [2600]
    [400345]    [5000]    [3500]    [2800]
    [400455]    [1200]    [ 900]    [ 800]
    [400876]    [3000]    [2400]    [1500]
    [400999]    [3000]    [1500]    [1000]


B =

    [400455]    [1200]    [ 900]    [ 800]
    [400999]    [3000]    [1500]    [1000]
    [400314]    [3000]    [2400]    [1800]
    [400876]    [3000]    [2400]    [1500]
    [400339]    [4300]    [ NaN]    [2600]
    [400345]    [5000]    [3500]    [2800]
```

For `B`, results are first sorted by January sales, in ascending order. The lowest value for January sales, 1200 (for item number 400455), appears first and the highest value, 5000 (for item number for 400345), appears last.

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

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

Creating Subqueries for Values from Multiple Tables

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

This example uses `basic.qry` (see "Creating and Running a Query to Import Data" on page 2-9). It retrieves sales volumes for the product whose description is Building Blocks. The table used for `basic.qry`, `salesVolume`, has sales volumes and a stock number field, but not a product description field. Another table, `productTable`, has the product description and stock number, but not the sales volumes. Therefore, the query needs to look at `productTable` to get the stock number for the product whose description is Building Blocks, and then has to look at the `salesVolume` table to get the sales volume values for that stock number:

1. Load `basic.qry`. For instructions, see “Using a Saved Query” on page 2-15.

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

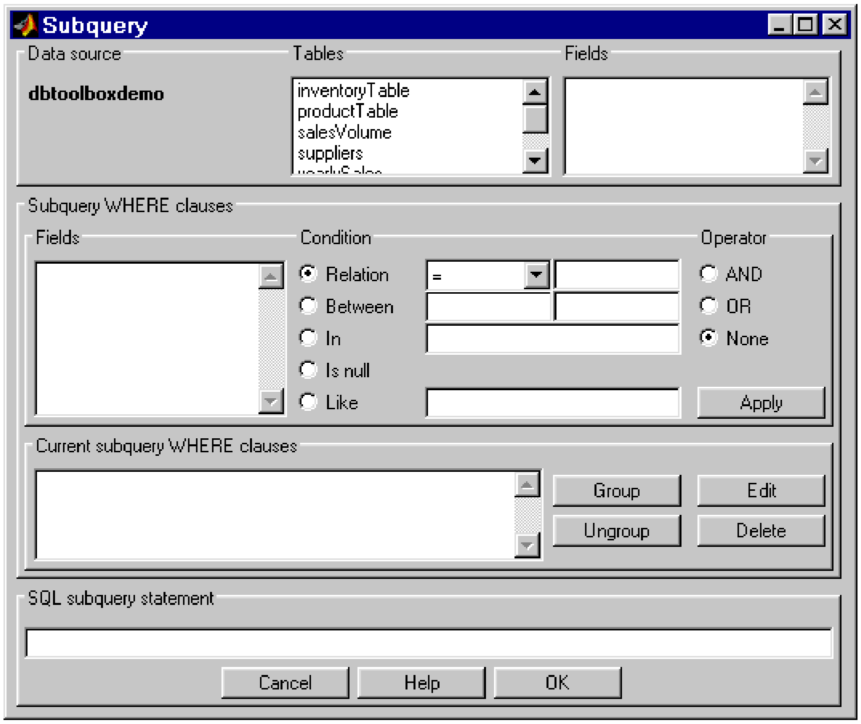
2. Set preferences. For this example, set **Data return format** to `cellarray` and **Read NULL numbers as** to `NaN`.

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

   The WHERE Clauses dialog box appears.

4. Click **Subquery**.

   The Subquery dialog box appears.



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

The fields in that table appear.

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

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

7 Create the condition that limits the query. In this example, limit the query to those product descriptions that are Building Blocks.

a In **Subquery WHERE clauses**, select `productDescription` from **Fields**.

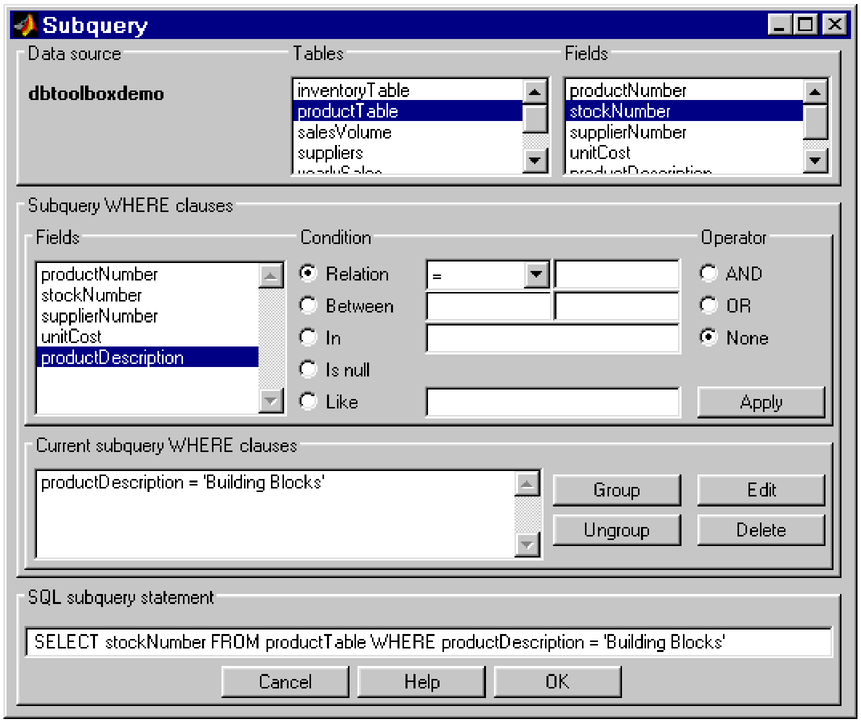
b For **Condition**, select **Relation**.

c From the drop-down list to the right of **Relation**, select =.

d In the field to the right of the drop-down list, type `'Building Blocks'` (include the single quotation marks to denote it is a string).

e Click **Apply**.

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



productDescription = 'Building Blocks'

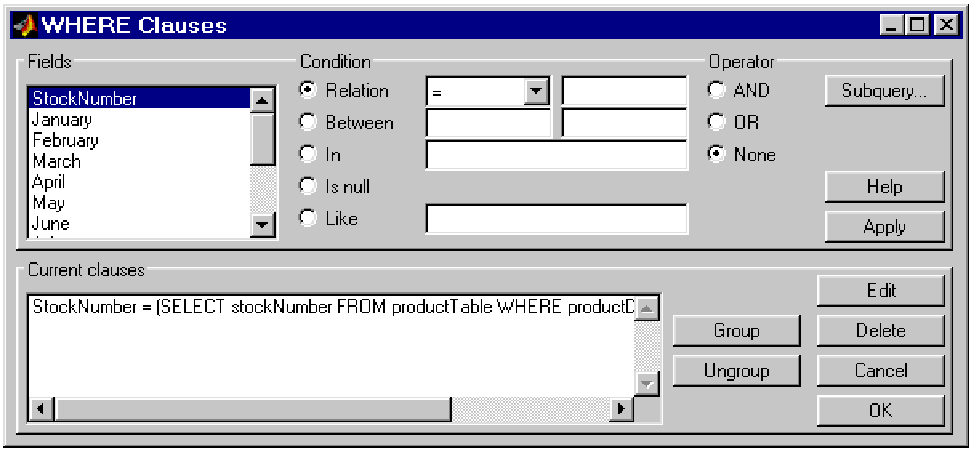
SELECT stockNumber FROM productTable WHERE productDescription = 'Building Blocks'

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

The Subquery dialog box closes.

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

This updates the **Current clauses** area using the subquery criteria specified in steps 3 through 8.



Fields: StockNumber, January, February, March, April, May, June

Condition: [✓ Relation] =, Between, In, Is null, Like

Operator: AND, OR, [✓ None]

Current clauses: StockNumber = (SELECT stockNumber FROM productTable WHERE productD

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

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

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

12 Click **Execute**.

The results are a 1-by-4 matrix.

13 Type `C` at the prompt in the Command Window to see the results.

```
C =

    [400345]    [5000]    [3500]    [2800]
```

14 The results are for item 400345, which has the product description Building Blocks, although that is not evident from the results. To verify that the product description is actually Building Blocks, run this simple query.

a Select `dbtoolboxdemo` as the **Data source**. This clears VQB selections made during a previous query.

b Select `productTable` from **Tables**.

c Select `stockNumber` and `productDescription` from **Fields**.

d Assign a **MATLAB workspace variable**, for example, `P`.

e Click **Execute**.

f Type `P` at the prompt in the Command Window to view the results.

```
P =

    [125970]    'Victorian Doll'
    [212569]    'Train Set'
    [389123]    'Engine Kit'
    [400314]    'Painting Set'
    [400339]    'Space Cruiser'
    [400345]    'Building Blocks'
    [400455]    'Tin Soldier'
    [400876]    'Sail Boat'
    [400999]    'Slinky'
    [888652]    'Teddy Bear'
```

The results show that item 400345 has the product description `Building Blocks.` "Creating Queries for Results from Multiple Tables" on page 2-52 creates a query that includes the product description in the results.

Creating Queries for Results from Multiple Tables

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

This example retrieves sales volumes by product description. The example is very similar to the example in "Creating Subqueries for Values from Multiple Tables" on page 2-47. 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 `salesVolume` table has a sales volume and a stock number field, but not a product description field. Another table, `productTable`, has the product description and the stock number, but not sales volumes. Therefore, the query needs to retrieve data from both tables and equate the stock number from `productTable` with the stock number from the `salesVolume` table:

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

2 For **Data operation**, choose **Select**.

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

   The **Catalog**, **Schema**, and **Tables** for that data source appear.

4 Do not specify **Catalog** and **Schema**. From **Tables**, select the tables from which you want to retrieve data. For example, **Ctrl**+click `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.

5 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`.

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

   The WHERE Clauses dialog box appears.

7 In the WHERE Clauses dialog box:

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

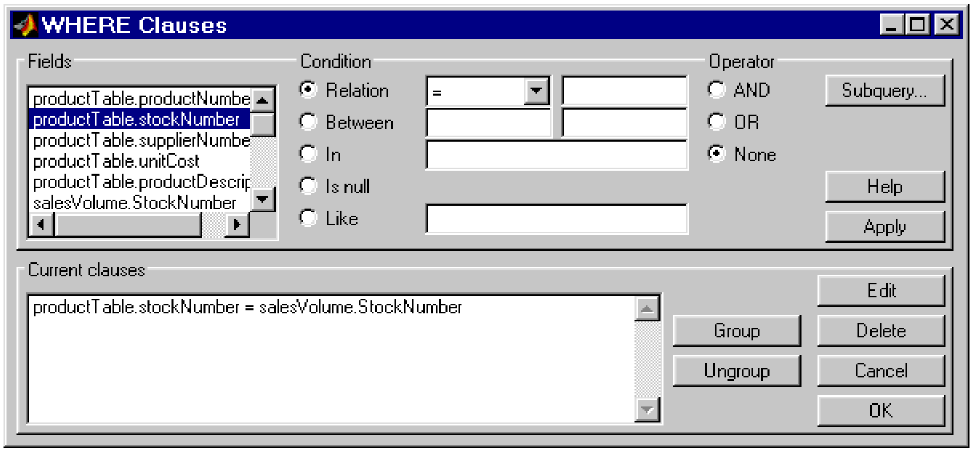
   b For **Condition**, select **Relation**.

   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.



Fields: productTable.productNumbe, productTable.stockNumber, productTable.supplierNumbe, productTable.unitCost, productTable.productDescrip, salesVolume.StockNumber

Condition: [✓ Relation] =, Between, In, Is null, Like

Operator: AND, OR, [✓ None]

Current clauses: productTable.stockNumber = salesVolume.StockNumber

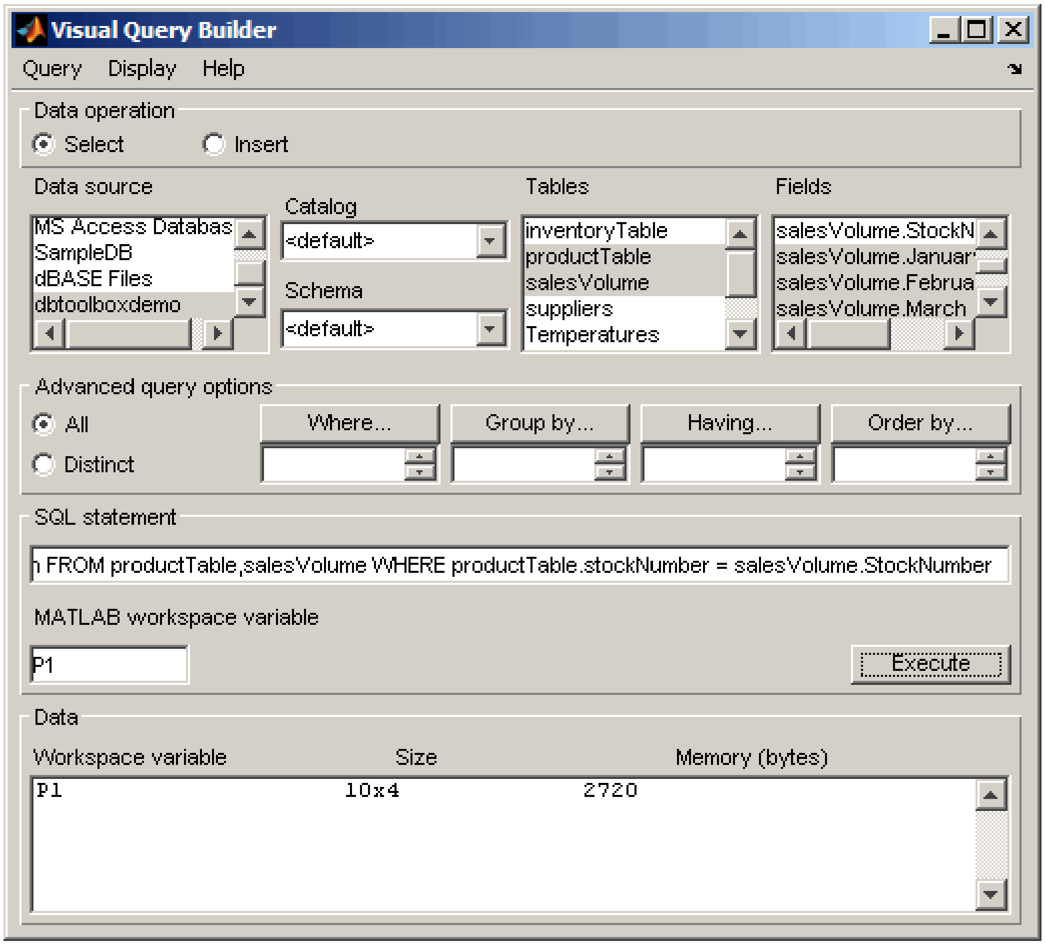
f Click **OK**.

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

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

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

The results are a 10-by-4 matrix.



Data operation: (●) Select ( ) Insert

Data source: MS Access Databas, SampleDB, dBASE Files, dbtoolboxdemo

Catalog: <default>

Schema: <default>

Tables: inventoryTable, productTable, salesVolume, suppliers, Temperatures

Fields: salesVolume.StockN, salesVolume.Januar, salesVolume.Februa, salesVolume.March

Advanced query options: (●) All ( ) Distinct

SQL statement:

```
n FROM productTable,salesVolume WHERE productTable.stockNumber = salesVolume.StockNumber
```

MATLAB workspace variable: P1

Workspace variable	Size	Memory (bytes)
P1	10x4	2720

10 Type P1 at the prompt in the Command Window to see the results.

```
P1 =

    'Victorian Doll'      [1400]     [1100]     [ 981]
    'Train Set'           [2400]     [1721]     [1414]
    'Engine Kit'          [1800]     [1200]     [ 890]
    'Painting Set'        [3000]     [2400]     [1800]
    'Space Cruiser'       [4300]     [ NaN]     [2600]
    'Building Blocks'     [5000]     [3500]     [2800]
```

```
'Tin Soldier'        [1200]    [ 900]    [ 800]
'Sail Boat'          [3000]    [2400]    [1500]
'Slinky'             [3000]    [1500]    [1000]
'Teddy Bear'         [ NaN]    [ 900]    [ 821]
```

Other Features in Advanced Query Options

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

Retrieving BINARY and OTHER Java Data

Database Toolbox supports the data types listed in “Data Types” on page 1-11, with no data manipulation required. You can also import `BINARY` and `OTHER` Java SQL objects, such as bitmap images. The process for importing `BINARY` and `OTHER` Java objects differs from the standard VQB import process in these ways:

- MATLAB cannot directly process these Java data types when retrieved. You need to understand the object contents to use the data. You might need to massage the data, such as stripping off leading entries added by your driver during data retrieval.
- For the `OTHER` data type, the returned data is sometimes empty because Java does not always successfully pass it through the JDBC/ODBC bridge.

Retrieving Images in Data

This example uses the `SampleDB` data source and a sample file for parsing image data, `*matlabroot*/toolbox/database/vqb/parsebinary.m`. For more information about the data source, see “Setting Up a Data Source” on page 1-13.

1 In the VQB dialog box, select

   a **Select** for **Data Operation**.

   b `SampleDB` from **Data source**.

   c `Employees` from **Tables**.

   d `EmployeeID` and `Photo` from **Fields**.

   The `Photo` field contains bitmap images.

2 Select **Query > Preferences** and specify the **Data return format** as `cellarray` or `structure`.

3 Assign `A` as the **MATLAB workspace variable** and click **Execute**.

4 Type A to view the contents.

MATLAB displays

```
A =

     [1]    [21626x1 int8]
     [2]    [21626x1 int8]
     [3]    [21722x1 int8]
     [4]    [21626x1 int8]
     [5]    [21626x1 int8]
     [6]    [21626x1 int8]
     [7]    [21626x1 int8]
     [8]    [21626x1 int8]
     [9]    [21626x1 int8]
```

5 Assign the first element in the image data to the variable photo. Type

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

6 Run the sample program parsebinary, which displays photo as a bitmap image. Because parsebinary outputs results, your current directory must be writable when you run it:

```
cd I:\MATLABFiles\myfiles
parsebinary(photo, 'BMP');
```

The bitmap image displays in a figure window.

The parsebinary M-file writes the retrieved data to a file, strips ODBC header information, and displays a bitmap image. For more details, type help parsebinary or view the parsebinary M-file in the MATLAB Editor/Debugger by typing open parsebinary.

This is just one example of retrieving a BINARY object. Your application might require different manipulations to process the data in MATLAB.

Exporting Data Using VQB

Build and run a query to export data from MATLAB into new rows in a database. Then save the query for use again later. The following sections describe how to do this.

- “Limitations” on page 2-59
- “Before You Start” on page 2-59
- “To Start” on page 2-59

Limitations

- You cannot use VQB to replace existing data in a database with data from MATLAB. Instead, use Database Toolbox `update` function.
- Use Database Toolbox functions instead of VQB if you use `commit` or `rollback` features when exporting data.
- Because VQB uses the `insert` function instead of `fastinsert`, you cannot export binary data using VQB, and data export operations are slower with VQB. You can instead use Database Toolbox `fastinsert` function to work around these limitations.

Before You Start

Before using VQB, set up a data source—see “Setting Up a Data Source” on page 1-13. The examples here use the `SampleDB` data source.

To Start

To open VQB, in the Command Window, type

```
querybuilder
```

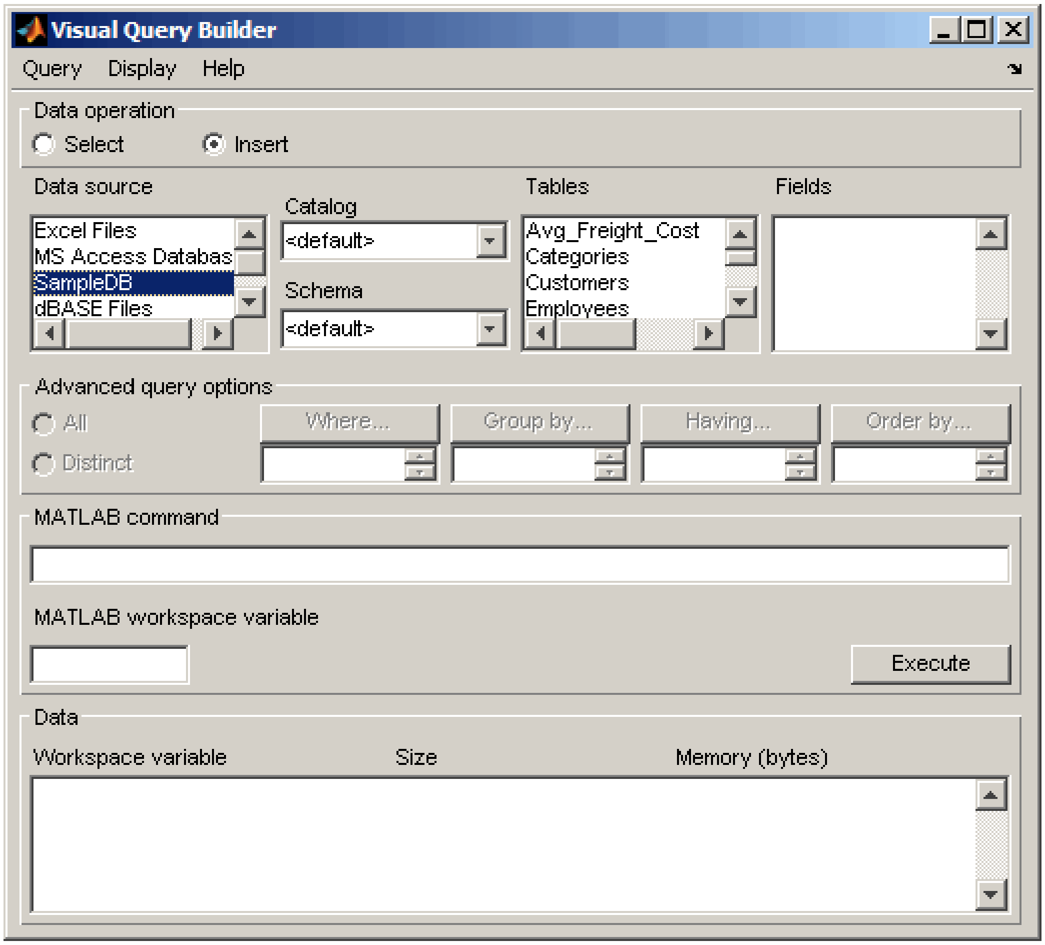
In VQB, perform these steps to create and run a query to export data:

1. In the **Data operation** field, select **Insert**, meaning you want to insert data into a database.

2 From the **Data source** list box, select the data source into which you want to export data. The list contains the data sources you defined in “Setting Up a Data Source” on page 1-13.

For this example, select `SampleDB`, which is the data source for the `Nwind` database.

After selecting a data source, the set of **Catalog**, **Schema**, and **Tables** in that data source appears.



3. Do not specify **Catalog** and **Schema**. From the **Tables** list box, select the table into which you want to export data. For this example, select `Avg_Freight_Cost`. Table names that include spaces appear in quotation marks. For a Microsoft Excel database, the **Tables** are Excel sheets.

   After you select a table, the set of **Fields** (column names) in that table appears.

4. From the **Fields** list box, select the fields into which you want to export data. To select more than one field, hold down the **Ctrl** key or **Shift** key while selecting. For this example, select the fields `Calc_Date` and `Avg_Cost`. Field names that include spaces appear in quotation marks. To deselect an entry, use **Ctrl**+click.

   As you select items from the **Fields** list, the query appears in the **MATLAB command** field.

5. Assign the data you want to export to a variable. For this example, type the following in the Command Window.

   ```
   export_data = {'07-Aug-2003',50.44};
   ```

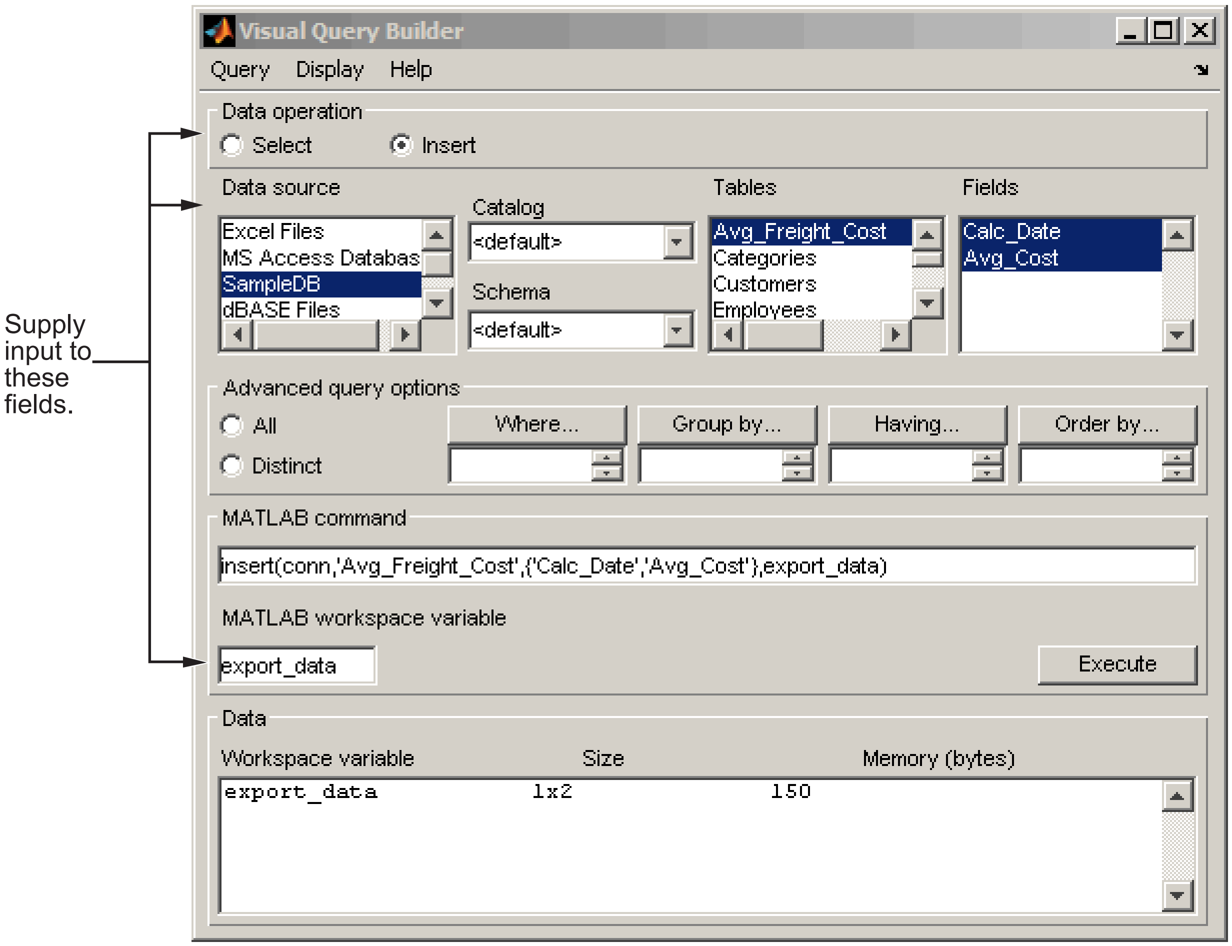
   This cell array contains a date and a numeric value.

   If the data contains `NULL` values, specify the format they take. Select **Query > Preferences** and specify **Write NULL numbers from** and **Write NULL strings from**. For more information about these preferences, see the property descriptions on the reference page for `setdbprefs`, which is the equivalent function for setting preferences.

6. In VQB **MATLAB workspace variable** field, enter the name of the variable whose data you want to export. For this example, use `export_data`. Press **Enter** or **Return** to view the **MATLAB command** that exports the data.

7. Click **Execute** to run the query and export the data.

   The query runs and exports the data. In the **Data** area, information about the exported data appears.



Data operation: Select [✓ Insert]

MATLAB command:

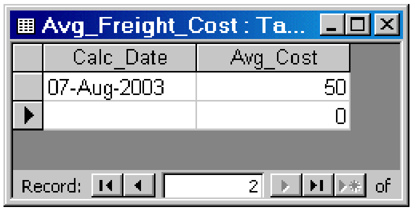
```
insert(conn,'Avg_Freight_Cost',{'Calc_Date','Avg_Cost'},export_data)
```

MATLAB workspace variable: export_data

```
export_data          1x2          150
```

If an error dialog box appears, the query is invalid. For example, you cannot export to a table or field name that contains quotation marks.

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



Calc_Date	Avg_Cost
07-Aug-2003	50
	0

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

9 To save this query, select **Query > Save** and name it `export.qry`. See "Saving, Editing, and Clearing Variables for Queries" on page 2-15. You can automatically generate an M-file that contains Database Toolbox functions to run this query—see "Generating M-Files from VQB Queries" on page 2-68.

BOOLEAN (MATLAB logical) Data

When you import data of the `BOOLEAN` type, MATLAB reads the data as a `logical` data type within the cell array or structure, having a value of `0` (false) or `1` (true). Similarly, you can export logical data from MATLAB to a database. This example illustrates both importing and exporting `BOOLEAN` data. For more information about the MATLAB `logical` data type, see "Logical Types" in the MATLAB Programming documentation.

- "Importing BOOLEAN Data" on page 2-64
- "Exporting BOOLEAN Data" on page 2-67

Importing BOOLEAN Data

1 Set preferences; for this example, set **Data return format** to `cellarray`.

2 For the **Data operation**, choose **Select**.

3 From **Data source**, select a data source; for this example, `SampleDB`.

4 From **Tables**, select a table; for this example, `Products`.

5 From **Fields**, select the fields; for this example, `ProductName` and `Discontinued`.

6 Assign the **MATLAB workspace variable**; for this example, use `D`.

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

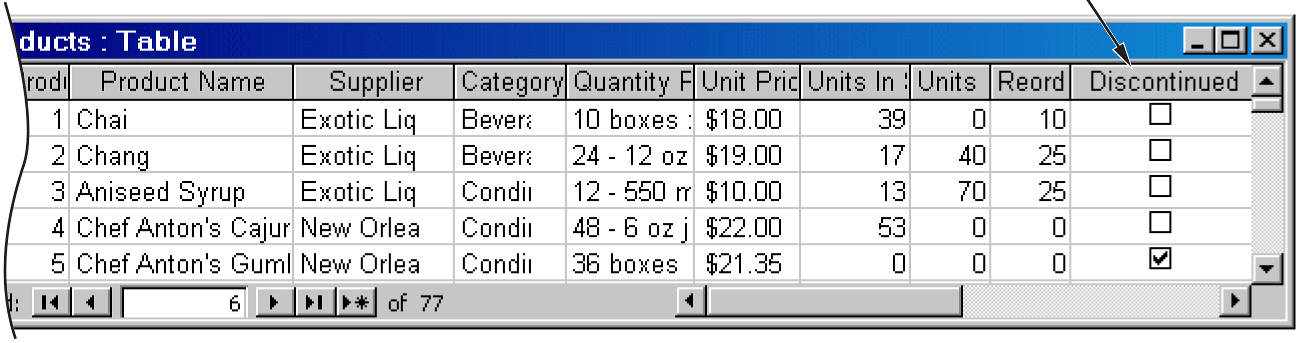
   VQB retrieves a 77-by-2 array.

8 Type `D` in the Command Window and MATLAB displays 77 records, with the first five shown here.

```
D =
    'Chai'                  [0]
    'Chang'                 [0]
    'Aniseed Syrup'         [0]
              [1x28 char]   [0]
              [1x22 char]   [1]
```

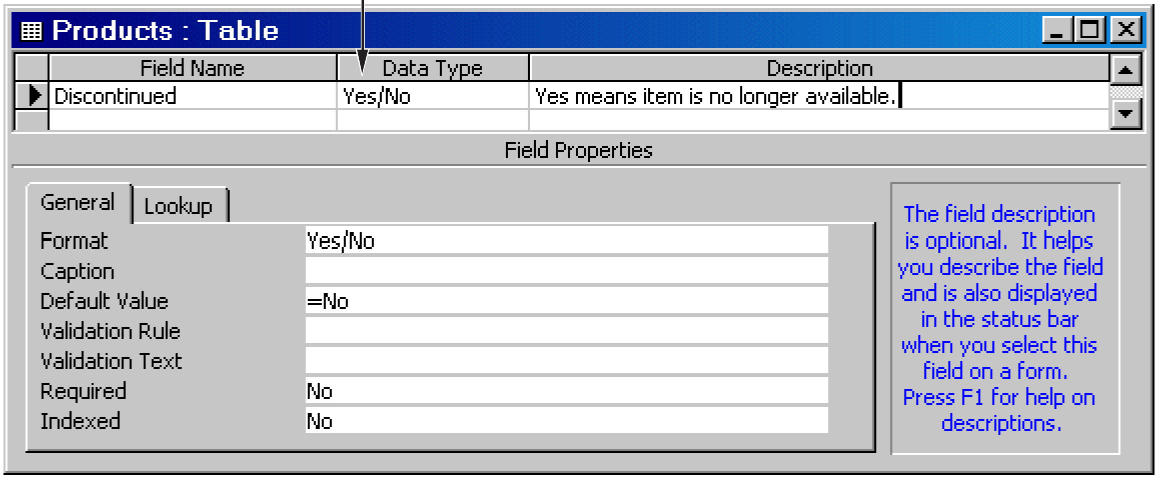
9 Compare this to the table in Microsoft Access.

Discontinued field is a BOOLEAN data type, where a check means true or Yes.



rodı	Product Name	Supplier	Category	Quantity F	Unit Pric	Units In	Units	Reord	Discontinued
1	Chai	Exotic Liq	Bever	10 boxes :	$18.00	39	0	10	☐
2	Chang	Exotic Liq	Bever	24 - 12 oz	$19.00	17	40	25	☐
3	Aniseed Syrup	Exotic Liq	Condi	12 - 550 m	$10.00	13	70	25	☐
4	Chef Anton's Cajur	New Orlea	Condi	48 - 6 oz j	$22.00	53	0	0	☐
5	Chef Anton's Guml	New Orlea	Condi	36 boxes	$21.35	0	0	0	☑

Design view in Access for the Discontinued field shows it is a Yes/No (BOOLEAN) data type.



Field Name	Data Type	Description
Discontinued	Yes/No	Yes means item is no longer available.

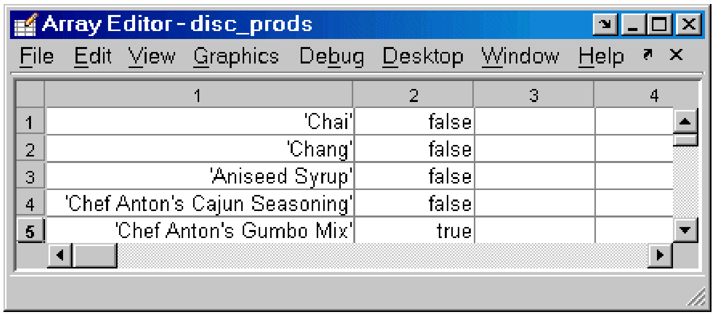
Field Properties

Format	Yes/No
Caption	
Default Value	=No
Validation Rule	
Validation Text	
Required	No
Indexed	No

The field description is optional. It helps you describe the field and is also displayed in the status bar when you select this field on a form. Press F1 for help on descriptions.

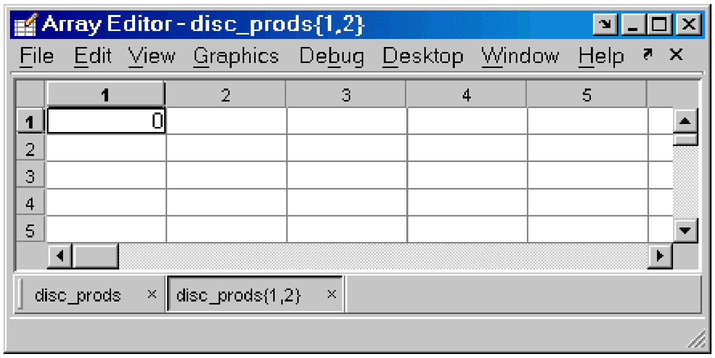
10 In the VQB Data area, double-click `D` to view the contents in the Array Editor.

The `logical` data type appears as `false` instead of `0` in the Array Editor cell array display. Double-click the `false` element in the cell array to view the logical value.


Array Editor - disc_prods
File Edit View Graphics Debug Desktop Window Help

	1	2	3	4
1	'Chai'	false		
2	'Chang'	false		
3	'Aniseed Syrup'	false		
4	'Chef Anton's Cajun Seasoning'	false		
5	'Chef Anton's Gumbo Mix'	true		

11 In the Array Editor, the logical value for the first product, `Chai`, appears as `false` instead of `0` for the cell array. This is to distinguish it as a `logical` value instead of a numeric `0`. In the Array Editor, double-click `false`. Its logical value, `0`, appears in a separate window.


Array Editor - disc_prods{1,2}
File Edit View Graphics Debug Desktop Window Help
disc_prods
disc_prods{1,2}

	1	2	3	4	5
1	0				
2					
3					
4					
5					

Exporting BOOLEAN Data

This example adds two rows of data to the `Products` table in the Access `Nwind` database.

1 In the MATLAB Command Window, create the structure `P`, which will be exported, by typing these commands:

```
P.ProductName{1,1}='Chocolate Truffles';
P.Discontinued{1,1}=logical(0);
P.ProductName{2,1}='Guatemalan Coffee';
P.Discontinued{2,1}=logical(1);
```

2 For the **Data operation**, choose **Insert**.

3 From **Data source**, select a data source; for this example, `SampleDB`.

4 From **Tables**, select a table; for this example, `Products`.

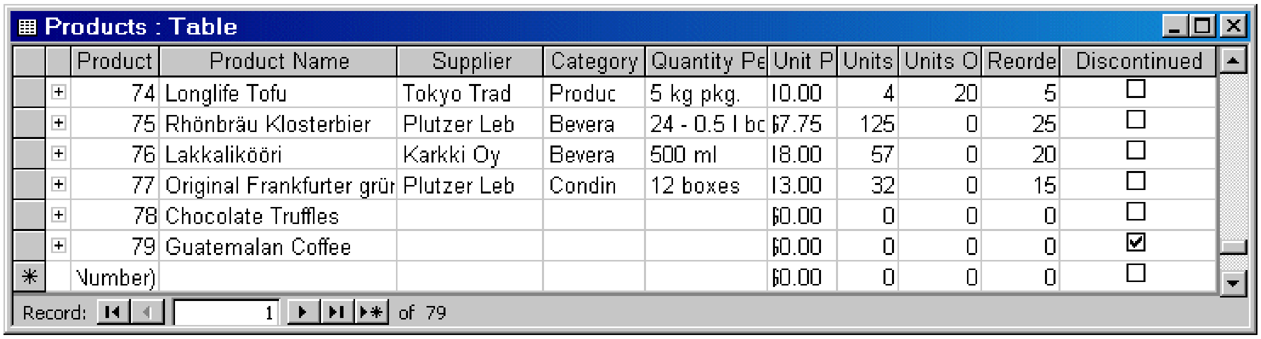
5 From **Fields**, select the fields; for this example, `ProductName` and `Discontinued`.

6 Assign the **MATLAB workspace variable**; for this example, use `P`.

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

VQB inserts two new rows into the `Products` table.

8 View the table in Microsoft Access to ensure the data was correctly inserted.



Products : Table

		Product	Product Name	Supplier	Category	Quantity Pe	Unit P	Units	Units O	Reorde	Discontinued
	+	74	Longlife Tofu	Tokyo Trad	Produc	5 kg pkg.	10.00	4	20	5	☐
	+	75	Rhönbräu Klosterbier	Plutzer Leb	Bevera	24 - 0.5 l bc	7.75	125	0	25	☐
	+	76	Lakkalikööri	Karkki Oy	Bevera	500 ml	18.00	57	0	20	☐
	+	77	Original Frankfurter grün	Plutzer Leb	Condin	12 boxes	13.00	32	0	15	☐
	+	78	Chocolate Truffles				0.00	0	0	0	☐
	+	79	Guatemalan Coffee				0.00	0	0	0	☑
*		Number)					0.00	0	0	0	☐

Record: 1 of 79

Generating M-Files from VQB Queries

The following topics are covered in this section:

- "About Generated M-Files" on page 2-68
- "Workspace Variable Not Included in Generated M-File" on page 2-69
- "Placeholder for Password Included in Generated M-File" on page 2-69

About Generated M-Files

Use Visual Query Builder to create a query easily via a GUI. Then select **Query > Generate M-File** to create a MATLAB M-file that contains Database Toolbox functions for that query.

You can then execute the M-file to run the query. You can also edit the M-file to include any MATLAB or related toolbox functions. Here is an example of a generated M-file:

```
% Set preferences with setdbprefs.
s.DataReturnFormat = 'cellarray';
s.ErrorHandling = 'store';
s.NullNumberRead = 'NaN';
s.NullNumberWrite = 'NaN';
s.NullStringRead = 'null';
s.NullStringWrite = 'null';
s.JDBCDataSourceFile = '';
s.UseRegistryForSources = 'yes';
s.TempDirForRegistryOutput = '';
setdbprefs(s)

% Make connection to database. Note that the password has been omitted.
% Using ODBC driver.
conn = database('dbtoolboxdemo','','password');

% Read data from database.
e = exec(conn,'SELECT ALL StockNumber,January,February FROM salesVolume');
e = fetch(e);
close(e)
```

```
% Close database connection.
close(conn)
```

Workspace Variable Not Included in Generated M-File

The M-file generated by VQB does not include the MATLAB workspace variable you provided when you created your query in VQB. The M-file assigns the results to `e`; you can access them after running the M-file using `e.Data`. You can also add a statement to the generated M-file assigning a variable name to `e.Data`, for example,

```
myVar = e.Data
```

Placeholder for Password Included in Generated M-File

For security reasons, the M-file generated by VQB does not include the actual password you used to connect to the database. Instead, the `database` statement includes the string `'password'` as a placeholder. For a database connection that does not require a password, the generated M-file will run as is. However, for database connections that require a password, substitute the actual password for the string `password` in the `database` statement so the M-file will run. Note that any database connection established via a JDBC driver requires a password.

Using Functions in Database Toolbox

When first using the toolbox, follow the simple examples in this section consecutively. Once you are familiar with the process, refer to the example of interest. To run these examples, you need to set up the specified data source—for instructions, see "Setting Up a Data Source" on page 1-13. If your version of Microsoft Access is different from the one used here, you might get different results. M-files containing functions used in some of these examples are in `matlab/toolbox/database/dbdemos`.

Accessing Metadata (p. 3-27)	Get information about the `dbtoolboxdemo` data source.
Performing Driver Functions (p. 3-34)	Create driver objects and set and get the properties (does not require you to set up a data source).
About Objects and Methods for Database Toolbox (p. 3-37)	Use object-oriented methods with Database Toolbox.
Working with Cell Arrays in MATLAB (p. 3-40)	Examples for the toolbox, if you are unfamiliar with cell arrays, used for mixed data types.

Importing Data into MATLAB from a Database

In this example, you connect to and import data from a database. Specifically, you connect to the `SampleDB` data source, and then import `country` data from the `customers` table in the `Nwind` sample database.

Note You can use the Visual Query Builder GUI instead of functions to import data from a database. See Chapter 2, "Visual Query Builder" for details.

In this section, you learn how to use these Database Toolbox functions:

- `database`
- `exec`
- `fetch` (`cursor.fetch`)
- `logintimeout`
- `ping`
- `setdbprefs`

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

1. If you did not already do so, set up the data source `SampleDB` according to the directions in "Setting Up a Data Source" on page 1-13.

2. In MATLAB, set the maximum time, in seconds, you want to allow the MATLAB session to try to connect to a database. This prevents the MATLAB session from hanging up if a database connection fails.

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

Type

```
logintimeout(5)
```

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

MATLAB returns

```
ans=
     5
```

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

3 Connect to the database by typing

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

- In this example, you define a MATLAB variable, `conn`, to be the returned connection object. This connection stays open until you close it with the `close` function.
- For the `database` function, you provide the name of the database, which is the data source `SampleDB` for this example. The other two arguments for the `database` function are `username` and `password`. For this example, they are empty strings because the `SampleDB` database does not require a username or password. To see a list of valid ODBC and JDBC data source names, run `getdatasources`.
- 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 via a structure.

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

4 Check the connection status by typing

```
ping(conn)
```

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

```
     DatabaseProductName: 'ACCESS'
  DatabaseProductVersion: '04.00.0000'
          JDBCDriverName: 'JDBC-ODBC Bridge (odbcjt32.dll)'
       JDBCDriverVersion: '2.0001 (04.00.6200)'
  MaxDatabaseConnections: 64
         CurrentUserName: 'admin'
             DatabaseURL: 'jdbc:odbc:SampleDB'
  AutoCommitTransactions: 'True'
```

5 Open a cursor and execute an SQL statement by typing

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

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

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

```
curs =
      Attributes: []
            Data: 0
  DatabaseObject: [1x1 database]
        RowLimit: 0
        SQLQuery: 'select country from customers'
         Message: []
            Type: 'Database Cursor Object'
       ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
          Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
       Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
           Fetch: 0
```

The data in the cursor object is stored in MATLAB.

If MATLAB displays an error, the query syntax might be invalid. See “Data Retrieval Restrictions” on page 1-8 for more information.

6 Specify the format of retrieved data by typing

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

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

7 Import data into MATLAB by typing

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

The fetch function imports data. It has the following two arguments in this example:

- curs, the cursor object returned by exec.
- 10, the maximum number of rows you want to be returned by fetch. The RowLimit argument is optional. If RowLimit is omitted, MATLAB

imports all remaining rows. When importing large quantities of data, rather than importing all the rows at once, import the data using multiple fetches with the `rowlimit` argument to improve speed and memory usage.

Note Database Toolbox has two forms of the `fetch` function (that is, two `fetch` methods) — `fetch` for a cursor object (`cursor.fetch`) as used here, and `fetch` for a connection object (`database.fetch`), which is a related convenience function. In either case, you use the syntax `fetch` along with the appropriate object argument—you do *not* explicitly specify `cursor.fetch` or `database.fetch`. Those references are provided to make the documentation explicit for each form of `fetch`. For more information about the use of `fetch`, `cursor.fetch`, and `database.fetch`, see `fetch`.

In this example, `fetch` reassigns the cursor object containing the rows of data returned by `fetch` to the variable `curs`. This is a best practice because it results in only one open cursor object, which means there is less memory usage, and you only have to close one cursor. 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 in turn contains the rows of data in the cell array. You can tell that `Data` contains 10 rows and 1 column.

The Attributes field is always empty. To view cursor attributes, you use the attr function, as described in “Viewing Information About the Imported Data” on page 3-10.

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

```
AA = curs.Data
```

MATLAB returns

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

Now you can use MATLAB to perform operations on the returned data. For more information, see "Working with Cell Arrays in MATLAB" on page 3-40. For more about working with strings, see "Characters and Strings" in the MATLAB Programming documentation.

- To import more rows of data, run the `fetch` statement from step 7 again and continue importing until all data is retrieved. At that point, `curs.Data` contains the string `'No Data'`.
- If the returned data includes Java `BINARY` or `OTHER` data types, you might have to process the data before using it. See "Retrieving BINARY or OTHER Java SQL Data Types" on page 3-25 for instructions to use this type of data.

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

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

Viewing Information About the Imported Data

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

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

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

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

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

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

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

MATLAB returns

```
numrows =
    10
```

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

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

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

MATLAB returns

```
numcols =
     1
```

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

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

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

MATLAB returns

```
colnames =
 'country'
```

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

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

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

MATLAB returns

```
colsize =
    15
```

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

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

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

MATLAB returns

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

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

7. Close the cursor by typing

   ```
   close(curs)
   ```

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

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

   ```
   close(conn)
   ```

Exporting Data from MATLAB to a New Record in a Database

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

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

Note You can use the Visual Query Builder GUI instead of functions to export data from MATLAB to new rows in a database. See Chapter 2, “Visual Query Builder” for details.

You learn to use these Database Toolbox functions:

- `get`
- `fastinsert`
- `setdbprefs`

1 If you are continuing from the previous example (“Viewing Information About the Imported Data” on page 3-10), skip this step. Otherwise, connect to the data source, `SampleDB`. Type

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

2 In MATLAB, set the format for retrieved data to `numeric` by typing

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

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

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

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

4 View the data you imported by typing

```
AA = curs.Data
```

MATLAB returns

```
AA =
   32.3800
   11.6100
   65.8300
```

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

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

MATLAB returns

```
meanA =
   36.6067
```

6 Assign the date on which this calculation was made to the variable `D` by typing

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

For more information about working with strings in MATLAB, see "Characters and Strings" in the MATLAB Programming documentation.

7 Assign the date and mean to a cell array, which you will export to the database. A cell array or structure is required because the date information is a string. Unlike importing data, you do not specify the export format

using `setdbprefs`, but instead use standard MATLAB operations to define it. Put the date in the first cell by typing

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

MATLAB returns

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

Put the mean in the second cell by typing

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

MATLAB returns

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

8 Define the names of the columns to which you will be exporting data. In this example, the column names are those in the `Avg_Freight_Cost` table you created earlier in "SampleDB Data Source" on page 1-14—`Calc_Date` and `Avg_Cost`. Assign the cell array containing the column names to the variable `colnames`. 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 by typing

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

MATLAB returns

```
ans =
 on
```

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

10 Export the data into the `Avg_Freight_Cost` table. For this example, type

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

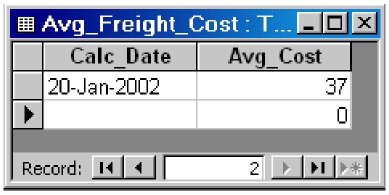
where `conn` is the connection object for the database to which you are exporting data. In this example, `conn` is `SampleDB`, which is already open. However, if you export to a different database that is not open, use the `database` function to connect to it before exporting the data. `Avg_Freight_Cost` is the name of the table to which you are exporting data. In the `fastinsert` function, you also include the `colnames` cell array and the cell array containing the data you are exporting, `exdata`, both of which you defined in the previous steps. Note that you do not define the type of data you are exporting; the data is exported in its current MATLAB format. Running `fastinsert` appends the data as a new record at the end of the `Avg_Freight_Cost` table.

If you get an error, it may be because the table is open in design mode in Access (edit mode for other databases). Close the table in Access and repeat the `fastinsert` function. For example, the error might be

```
[Vendor][ODBC Product Driver] The database engine could not
lock table 'TableName' because it is already in use by
another person or process.
```

If you have other problems using `fastinsert`, try using `insert` instead.

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



Calc_Date	Avg_Cost
20-Jan-2002	37
	0

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 by typing

```
close(curs)
```

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

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

```
close(conn)
```

Do not delete or change the `Avg_Freight_Cost` table in Access because you will use it in the next example.

Replacing Existing Data in a Database from MATLAB

In this example, you update existing data in the database with exported data from MATLAB. Specifically, you update the date you previously imported into the `Avg_Freight_Cost` table.

You learn to use these Database Toolbox functions:

- `close`
- `update`

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

1 If you are continuing directly from the previous example ("Exporting Data from MATLAB to a New Record in a Database" on page 3-13), skip this step. Otherwise, type

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

MATLAB returns

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

2 Assume that the date in the `Avg_Freight_Cost` table is incorrect and instead should be `19-Jan-2002`. Type

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

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

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

MATLAB returns

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

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

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

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

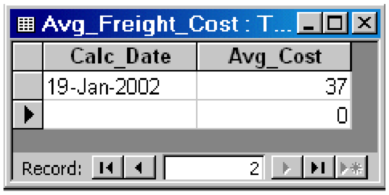
```
whereclause =
 where Calc_Date = '20-Jan-2002'
```

For more information about working with strings in MATLAB, see "Characters and Strings" in the MATLAB Programming documentation.

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

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

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


Avg_Freight_Cost : T...

Calc_Date	Avg_Cost
19-Jan-2002	37
	0

Record: 2

7 Close the cursor and 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 New Records from MATLAB

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

You learn to use these Database Toolbox functions:

- `fastinsert`
- `setdbprefs`

If you want to see or copy the functions for this example, or if you want to run a similar set of functions, use the M-file `matlab\toolbox\database\dbdemos\dbinsert2demo.m`.

1 If you did not already do so, set up the data source `dbtoolboxdemo` according to the directions in "Setting Up a Data Source" on page 1-13. This data source uses the `tutorial` database.

2 Check the properties of the `tutorial` database to be sure it is writable, that is, *not* read only.

3 Connect to the database by typing

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

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

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

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

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

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

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

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

```
columnnames(curs)
```

MATLAB returns

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

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

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

MATLAB returns

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

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

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

MATLAB returns

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

9 Compute the monthly totals by typing

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

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

For example, when `c` is `2`, row `1` of `monthly` is the total of all rows in column `2` of `curs.Data`, where column `2` is the sales volume for January.

To see the result, type

```
monthly
```

MATLAB returns

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

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

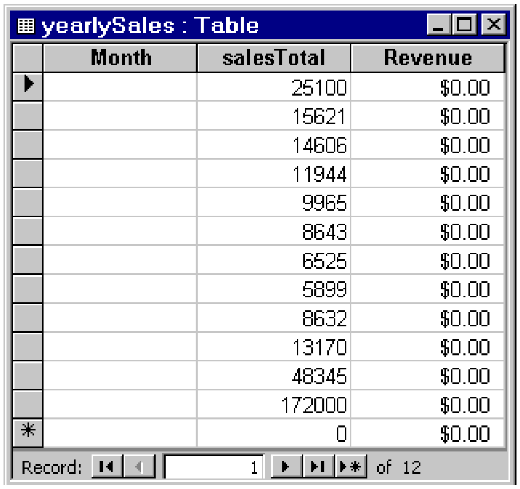
```
colnames{1,1} = 'salesTotal';
```

11 Insert the data into the `yearlySales` table by typing

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

Be sure the database properties are not read only or archive.

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



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

13 Close the cursor and database connection. Type

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

Retrieving BINARY or OTHER Java SQL Data Types

You can retrieve `BINARY` or `OTHER` Java SQL data types; however, the data might require additional processing once retrieved. For example, you can retrieve data from a MAT-file or an image file. MATLAB cannot process these data types directly. You need knowledge of the content and might need to massage the data in order to work with it in MATLAB, such as stripping off leading entries added by your driver during data retrieval.

In this example, you import data that includes bitmap images. You use a sample M-file included with Database Toolbox in the `vqb` directory:

- `parsebinary`

Perform these steps to retrieve bitmap image data for the example:

1. Connect to the data source, `SampleDB`. Type

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

2. For the data return format preference, specify either `cellarray` or `structure`. For this example, set it to `cellarray` by typing

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

3. Import the data, which includes bitmap image files. For the example, import the `EmployeeID` and `Photo` columns of data from the `Employees` table. Type

   ```
   curs = exec(conn, 'select EmployeeID,Photo from Employees')
   curs = fetch(curs);
   ```

4. View the data you imported by typing

   ```
   curs.Data
   ```

MATLAB returns

```
ans = 

    [1]    [21626x1 int8]
    [2]    [21626x1 int8]
    [3]    [21722x1 int8]
    [4]    [21626x1 int8]
    [5]    [21626x1 int8]
    [6]    [21626x1 int8]
    [7]    [21626x1 int8]
    [8]    [21626x1 int8]
    [9]    [21626x1 int8]
```

Some of the `OTHER` data type fields might be empty. This happens when Java cannot pass the data through the JDBC/ODBC bridge, for example.

5 Assign the image element you want to the variable `photo`. Type

```
photo = curs.Data{1,2};
```

6 Run the sample program, `*matlabroot*/toolbox/database/vqb/parsebinary.m`, which displays `photo` as a bitmap image. Because `parsebinary` outputs results, your current directory must be writable when you run it:

```
cd 'I:\MATLABFiles\myfiles
parsebinary(photo, 'BMP');
```

The bitmap image displays in a figure window. The `parsebinary` M-file writes the retrieved data to a file, strips ODBC header information, and displays a bitmap image. For more details, type `help parsebinary` or view the `parsebinary` M-file in the MATLAB Editor/Debugger by typing `open parsebinary`.

This is just one example of retrieving a `BINARY` or `OTHER` object. Your application might require different manipulations to process the data in MATLAB.

Accessing Metadata

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

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

1 Connect to the `dbtoolboxdemo` data source. Type

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

MATLAB returns information about the database object.

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

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

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

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

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

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

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

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

```
v =
              AllProceduresAreCallable: 1
                AllTablesAreSelectable: 1
 DataDefinitionCausesTransactionCommit: 1
   DataDefinitionIgnoredInTransactions: 0
            DoesMaxRowSizeIncludeBlobs: 0
                              Catalogs: {4x1 cell}
                      CatalogSeparator: '.'
                           CatalogTerm: 'DATABASE'
                   DatabaseProductName: 'ACCESS'
                DatabaseProductVersion: '04.00.0000'
           DefaultTransactionIsolation: 2
                    DriverMajorVersion: 2
                    DriverMinorVersion: 1
                            DriverName: [1x31 char]
                         DriverVersion: '2.0001 (04.00.6200)'
                   ExtraNameCharacters: [1x29 char]
                 IdentifierQuoteString: '`'
                      IsCatalogAtStart: 1
                MaxBinaryLiteralLength: 255
                  MaxCatalogNameLength: 260
                  MaxCharLiteralLength: 255
                   MaxColumnNameLength: 64
                   MaxColumnsInGroupBy: 10
                     MaxColumnsInIndex: 10
                   MaxColumnsInOrderBy: 10
                    MaxColumnsInSelect: 255
                     MaxColumnsInTable: 255
                        MaxConnections: 64
                   MaxCursorNameLength: 64
                        MaxIndexLength: 255
                MaxProcedureNameLength: 64
                            MaxRowSize: 4052
                   MaxSchemaNameLength: 0
                    MaxStatementLength: 65000
                         MaxStatements: 0
                 MaxTableNameLength: 64
                  MaxTablesInSelect: 16
                  MaxUserNameLength: 0
                   NumericFunctions: [1x73 char]
                      ProcedureTerm: 'QUERY'
                            Schemas: {}
                         SchemaTerm: ''
                 SearchStringEscape: '\'
                        SQLKeywords: [1x461 char]
                    StringFunctions: [1x91 char]
         StoresLowerCaseIdentifiers: 0
   StoresLowerCaseQuotedIdentifiers: 0
         StoresMixedCaseIdentifiers: 0
   StoresMixedCaseQuotedIdentifiers: 1
         StoresUpperCaseIdentifiers: 0
   StoresUpperCaseQuotedIdentifiers: 0
                    SystemFunctions: ''
                         TableTypes: {13x1 cell}
                  TimeDateFunctions: [1x111 char]
                           TypeInfo: {16x1 cell}
                                URL: 'jdbc:odbc:dbtoolboxdemo'
                           UserName: 'admin'
               NullPlusNonNullIsNull: 0
                 NullsAreSortedAtEnd: 0
               NullsAreSortedAtStart: 0
                  NullsAreSortedHigh: 0
                   NullsAreSortedLow: 1
               UsesLocalFilePerTable: 0
                      UsesLocalFiles: 1
```

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

```
v.Catalogs
```

MATLAB returns

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

For more information about the database metadata properties returned by `get`, see the methods of the `DatabaseMetaData` object at the Java Web site.

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

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

MATLAB returns

```
a =
                                 AlterTableWithAddColumn: 1
                                AlterTableWithDropColumn: 1
                                     ANSI92EntryLevelSQL: 1
                                           ANSI92FullSQL: 0
                                   ANSI92IntermediateSQL: 0
                              CatalogsInDataManipulation: 1
                              CatalogsInIndexDefinitions: 1
                          CatalogsInPrivilegeDefinitions: 0
                                CatalogsInProcedureCalls: 0
                              CatalogsInTableDefinitions: 1
                                          ColumnAliasing: 1
                                                 Convert: 1
                                          CoreSQLGrammar: 0
                                    CorrelatedSubqueries: 1
        DataDefinitionAndDataManipulationTransactions: 1
```

```
      DataManipulationTransactionsOnly: 0
        DifferentTableCorrelationNames: 0
                   ExpressionsInOrderBy: 1
                     ExtendedSQLGrammar: 0
                         FullOuterJoins: 0
                                GroupBy: 1
                    GroupByBeyondSelect: 1
                       GroupByUnrelated: 0
          IntegrityEnhancementFacility: 0
                       LikeEscapeClause: 0
                      LimitedOuterJoins: 0
                      MinimumSQLGrammar: 1
                   MixedCaseIdentifiers: 1
             MixedCaseQuotedIdentifiers: 0
                     MultipleResultSets: 0
                   MultipleTransactions: 1
                     NonNullableColumns: 0
                OpenCursorsAcrossCommit: 0
              OpenCursorsAcrossRollback: 0
             OpenStatementsAcrossCommit: 1
           OpenStatementsAcrossRollback: 1
                       OrderByUnrelated: 0
                             OuterJoins: 1
                       PositionedDelete: 0
                       PositionedUpdate: 0
              SchemasInDataManipulation: 0
              SchemasInIndexDefinitions: 0
          SchemasInPrivilegeDefinitions: 0
                SchemasInProcedureCalls: 0
              SchemasInTableDefinitions: 0
                        SelectForUpdate: 0
                       StoredProcedures: 1
                SubqueriesInComparisons: 1
                     SubqueriesInExists: 1
                        SubqueriesInIns: 1
                SubqueriesInQuantifieds: 1
                  TableCorrelationNames: 1
                           Transactions: 1
                                  Union: 1
                               UnionAll: 1
```

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

For more information about the properties supported by the database, see the methods of the DatabaseMetaData object at the Java Web site.

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

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

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

MATLAB returns the names and types for each table.

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

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

For a list of all of the database metadata functions, see "Database Metadata Object" on page 4-4. 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` (`cursor.fetch`). You can then view properties of the resultset object using `get`, create a resultset metadata object using `rsmd` and get its properties, or make calls to the resultset object using your own Java-based applications. For more information, see the reference pages for `resultset` and `rsmd`, or see the lists of related functions, "Resultset Object" on page 4-6 and "Resultset Metadata Object" on page 4-6.

Performing Driver Functions

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

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

Note There is no equivalent M-file demo to run because the example relies on a specific system-to-JDBC connection and database. Your configuration will be different from the one in this example, so you cannot run these examples exactly as written. Instead, use values for your own system. See your database administrator for address information.

---

1 Connect to the database.

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

2 Use the `driver` function to construct a driver object for a specified database URL string of the form `jdbc:`*`subprotocol`*`:`*`subname`*. For example, type

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

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

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

3 To get properties of the driver object, type

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

MATLAB returns information about the driver's versions.

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

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

```
isdriver(d)
```

MATLAB returns

```
ans =
 1
```

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

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

```
dm = drivermanager
```

`dm` is the drivermanager object.

6 Get properties of the drivermanager object. Type

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

MATLAB returns

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

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

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

Verify the value by typing

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

MATLAB returns

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

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

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

MATLAB returns

```
ans =
 10
```

For a list of all the driver object functions, see "Driver Object" on page 4-5 and "Drivermanager Object" on page 4-6.

About Objects and Methods for Database Toolbox

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 Database Toolbox with no knowledge of or interest in its object-oriented implementation. But for those who are interested, some of its useful characteristics follow:

- You use constructor functions to create objects, such as running the `fetch` (`cursor.fetch`) function to create a cursor object containing query results. MATLAB returns not only the object but also the stored information about the object. Because 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 Database Toolbox, you use the `get` function, regardless of the object. This means you have to remember only 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 Database Toolbox. For more information, see "Classes and Objects" in the MATLAB documentation.

Working with Cell Arrays in MATLAB

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

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

- “Viewing Cell Array Data Returned from a Query” on page 3-40
- “Viewing Elements of Cell Array Data” on page 3-43
- “Performing Functions on Cell Array Data” on page 3-45
- “Creating Cell Arrays for Exporting Data from MATLAB” on page 3-45

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

You can use structures instead of cell arrays. For more information, see “Structures” in the MATLAB Programming documentation.

You also might also need more information about working with strings in MATLAB. See the functions `char`, `cellstr`, and `strvcat` and “Characters and Strings” in the MATLAB Programming documentation.

Viewing Cell Array Data Returned from a Query

Viewing Query Results

How you view query results depends on if you import the data using the `fetch` (`cursor.fetch`) function or if you use Visual Query Builder.

Using the fetch Function. If you import data using the `fetch` function (`cursor.fetch`), MATLAB returns data to a cursor object, as in the following data, which was imported in the example "Exporting Data from MATLAB to a New Record in a Database" on page 3-13.

```
curs =
        Attributes: []
              Data: [3x1 double]
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: 'select freight from orders'
           Message: []
              Type: 'Database Cursor Object'
         ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
            Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
         Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
             Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

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

```
curs.Data
```

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

```
A = curs.Data
```

and then view it by typing `A`.

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

MATLAB displays the data in the Command Window, for example

```
A =
    32.3800
    11.6100
    65.8300
```

Viewing Results with Multiple Columns

If the query results consist of multiple columns, you can view all the results for a single column using a colon (:). See the example in “Exporting Multiple New Records from MATLAB” on page 3-20. For example, you view the results of column 2 by typing

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

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

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

MATLAB returns the data in column 2, for example

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

Expanding Results

If the results do not fit in the display space available, MATLAB displays size information only. If, for example, MATLAB returns these query results.

```
B =
    [1]    'Beverages'      [1x43 char]
    [2]    'Condiments'     [1x58 char]
    [3]    'Confections'    [1x35 char]
```

You can see the data in columns 1 and 2, but the third is expressed as an array because the results are too long to display.

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

```
B{1,3}
```

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

```
curs.Data{1,3}
```

MATLAB returns

```
ans =
 Soft drinks, coffees, teas, beers, and ales
```

Viewing Elements of Cell Array Data

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

This example is the same as that in "Exporting Data from MATLAB to a New Record in a Database" on page 3-13, but the `DataReturnFormat` is set to `cellarray`.

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

Viewing a Single Element as a Numeric Value

To view the first element of `A`, type

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

MATLAB returns

```
ans =
 [32.3800]
```

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

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

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

MATLAB returns

```
ans =
 32.3800
```

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

Viewing an Entire Column or Row as a Numeric Vector

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

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

MATLAB returns

```
AA =
   32.3800
   11.6100
   65.8300
```

You can also use the contents with the `celldisp` function. For example, type

```
celldisp(A)
```

MATLAB returns

```
A{1} =
 32.3800

A{2} =
 11.6100

A{3} =
 65.8300
```

Performing Functions on Cell Array Data

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

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

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

MATLAB returns

```
ans =
 109.8200
```

Creating Cell Arrays for Exporting Data from MATLAB

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

Enclosing Data in Curly Braces

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

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

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

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

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

Assigning Cell Array Elements

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

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

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

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

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

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

Converting a Numeric Matrix to a Cell Array

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

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

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

Functions — By Category

General

`logintimeout`	Set or get time allowed to establish database connection
`setdbprefs`	Set preferences for retrieval format, errors, `NULL`s, and more

Database Connection

`close`	Close database connection, cursor, or resultset object
`database`	Connect to database
`get`	Object properties
`getdatasources`	Names of valid ODBC and JDBC data sources on system
`isconnection`	Detect whether database connection is valid
`isreadonly`	Detect whether database connection is read only
`ping`	Status information about database connection
`set`	Set properties for database, cursor, or drivermanager object
`setdbprefs`	Set preferences for retrieval format, errors, `NULL`s, and more
`sql2native`	Convert JDBC SQL grammar to system's native SQL grammar

SQL Cursor

`close`	Close database connection, cursor, or resultset object
`exec`	Execute SQL statement and open cursor
`get`	Object properties
`querytimeout`	Time allowed for database SQL query to succeed
`runstoredprocedure`	Call stored procedure with input and output parameters
`set`	Set properties for database, cursor, or drivermanager object

Importing Data into MATLAB from a Database

`attr`	Attributes of columns in fetched data set
`cols`	Number of columns in fetched data set
`columnnames`	Names of columns in fetched data set
`cursor.fetch`	Import data into MATLAB from cursor object created by `exec`
`database.fetch`	Execute SQL statement and import data into MATLAB
`fetch`	`cursor.fetch` or `database.fetch`
`fetchmulti`	Import data into MATLAB from multiple resultsets
`querybuilder`	Start SQL query builder GUI to import and export data

`rows`	Number of rows in fetched data set
`width`	Field size of column in fetched data set

Database Metadata Object

`bestrowid`	Database table unique row identifier
`columnprivileges`	Database column privileges
`columns`	Database table column names
`crossreference`	Information about primary and foreign keys
`dmd`	Construct database metadata object
`exportedkeys`	Information about exported foreign keys
`get`	Object properties
`importedkeys`	Information about imported foreign keys
`indexinfo`	Indices and statistics for database table
`primarykeys`	Primary key information for database table or schema
`procedurecolumns`	Catalog’s stored procedure parameters and result columns
`procedures`	Catalog’s stored procedures
`supports`	Detect whether property is supported by database metadata object
`tableprivileges`	Database table privileges

`tables`	Database table names
`versioncolumns`	Automatically updated table columns

Exporting Data from MATLAB to a Database

`commit`	Make database changes permanent
`insert`	Add MATLAB data to database table (deprecated; use `fastinsert` instead)
`querybuilder`	Start SQL query builder GUI to import and export data
`rollback`	Undo database changes
`update`	Replace data in database table with data from MATLAB

Driver Object

`driver`	Construct database driver object
`get`	Object properties
`isdriver`	Detect whether driver is valid JDBC driver object
`isjdbc`	Detect whether driver is JDBC compliant
`isurl`	Detect whether database URL is valid
`register`	Load database driver
`unregister`	Unload database driver

Drivermanager Object

`drivermanager`	Construct database drivermanager object
`get`	Object properties
`set`	Set properties for database, cursor, or drivermanager object

Resultset Object

`clearwarnings`	Clear warnings for database connection or resultset
`close`	Close database connection, cursor, or resultset object
`get`	Object properties
`isnullcolumn`	Detect whether last record read in resultset was `NULL`
`namecolumn`	Map resultset column name to resultset column index
`resultset`	Construct resultset object

Resultset Metadata Object

`get`	Object properties
`rsmd`	Construct resultset metadata object

Visual Query Builder

`confds`	Configure data source for Visual Query Builder (JDBC)
`querybuilder`	Start SQL query builder GUI to import and export data

Functions — Alphabetical List

Purpose

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`, `cursor.fetch`,`dmd`, `get`, `tables`, `width`

Purpose

Database table unique row identifier

Syntax

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

Description

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

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

Examples

Type

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

MATLAB returns

```
b =
      'build_id'
```

In this example

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

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

See Also

`columns`, `dmd`, `get`, `tables`

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)` clears 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
```

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

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

See Also `cursor.fetch`, `database`, `exec`, `resultset`

Purpose

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`, `cursor.fetch`, `get`, `rows`, `width`

Purpose

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`, `cursor.fetch`, `get`, `width`

Purpose

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 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`.

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

Examples

Type

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

MATLAB returns

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

In this example

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

The results show

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

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

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

MATLAB returns the column privileges for the build_id column.

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

See Also cols, columns, columnnames, dmd, get

Purpose

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 `fastinsert`, `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

```
fastinsert(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`, `fastinsert`, `get`, `rollback`, `update`

confds

Purpose Configure data source for Visual Query Builder (JDBC)

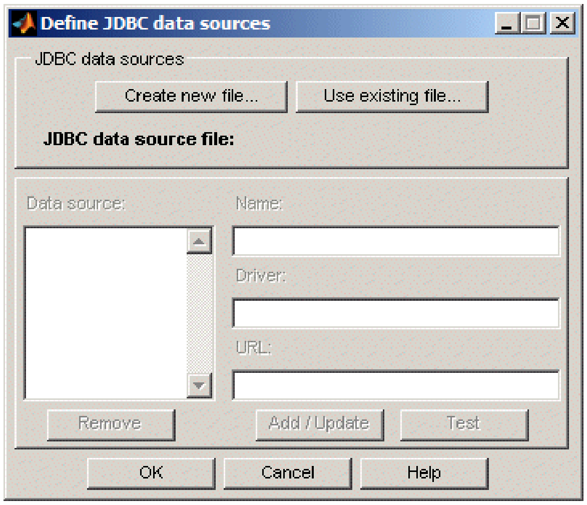
GUI Alternatives As an alternative to the `confds` function, you can select **Define JDBC data sources** from the Visual Query Builder **Query** menu.

Syntax

```
confds
```

Description `confds` displays the Define JDBC Data Sources dialog box, with which you add and remove data sources for use with Visual Query Builder (VQB). Use `confds` only if you want to build and run queries using Visual Query Builder via JDBC drivers.

To use JDBC data sources with Database Toolbox functions, you instead define the JDBC data source when you establish the connection using the `database` function. To add and remove data sources for connections that use ODBC drivers, see "Setting Up a Data Source" on page 1-13.



To use a data source with JDBC drivers, you must include a reference that specifies the location of the JDBC drivers file in a MATLAB Java classpath file. Then complete the Define JDBC Data Sources dialog box by performing these steps:

1 "Find Your JDBC Drivers Filename" on page 1-20.

2 "Include the Reference in the MATLAB Java Classpath" on page 1-21.

3 "Define a JDBC Data Source in Visual Query Builder" on page 1-23 (skip to step 2 in those instructions).

See Also

`database` (for examples of JDBC drivers and URLs), `querybuilder`

crossreference

Purpose

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 the Java Web site for the `getCrossReference` property.

See Also `dmd`, `exportedkeys`, `get`, `importedkeys`, `primarykeys`

Purpose

Import data into MATLAB from cursor object created by `exec`

GUI Alternatives

As an alternative to the `fetch` function, you can retrieve data using Visual Query Builder. Run `querybuilder` and use the **Help** menu for more information.

Syntax

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

Description

`curs = fetch(curs, RowLimit)` imports rows of data from the open SQL cursor `curs` (created using `exec`), up to the maximum `RowLimit`, into the object `curs`. Data is stored in MATLAB in a cell array, structure, or numeric matrix, based on specifications made using `setdbprefs`. It is best practice to assign the object returned by `fetch` to the variable `curs` from the open SQL cursor. This practice results in only one open cursor object, which means there is less memory usage, and you only have to close one cursor. The next time you run `fetch`, records are imported starting with the row following `RowLimit`. If you fetch large amounts of data that cause memory or speed problems, use `RowLimit` to limit how much data is retrieved at once.

`curs = fetch(curs)` imports rows of data from the open SQL cursor `curs`, up to the `RowLimit` specified by `set`, into the object `curs`. Data is stored in MATLAB in a cell array, structure, or numeric matrix, based on specifications you made using `setdbprefs`. It is a best practice to assign the object returned by `fetch` to the variable `curs` from the open SQL cursor. This practice results in only one open cursor object, which means there is less memory usage, and you only have to close one cursor. 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.

Remarks

Note This page documents `fetch` for a cursor object. For more information about the use of `fetch`, `cursor.fetch`, and `database.fetch`, see `fetch`. Unless otherwise noted, `fetch` in this documentation refers to `cursor.fetch`, rather than `database.fetch`.

- Do not count on the order of records in your database as being constant, but rather always use the values in column names to identify records. You can use the SQL `ORDER BY` command in your `exec` statement to sort the data.
- Running `fetch` returns information about the cursor object, `curs`, created using `exec`. The `Data` element of the cursor object contains the data returned by `fetch`. The data types are preserved. After running `fetch`, display the returned data by typing `curs.Data`.
- You can only retrieve a single resultset using `fetch`. To retrieve multiple resultsets, use `fetchmulti`.
- When a fetched field contains `BOOLEAN` data, it is represented as a `logical` data type in MATLAB.
- When a field in `curs.Data` contains `BINARY` or `OTHER` data types, you might need to understand the content and process it before using it in MATLAB. See "Retrieving BINARY or OTHER Java SQL Data Types" on page 3-25 for a specific example about processing bitmap image data.
- Use `get` to view properties of `curs`, and `attr` to view cursor attributes.

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 contained in the cursor object field `curs.Data`. To display data in `curs.Data`, type

```
curs.Data
```

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

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

Example 2 — Import Specified Number of Rows of Data

Specify the `RowLimit` argument to retrieve the first three 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 three rows of data. Adding the semicolon suppresses display of the results.

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

Display the data by typing

```
curs.Data
```

MATLAB returns

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

Example 3 — Repeat Importing Rows to Retrieve All Data

In this example, specify the `RowLimit` argument to retrieve the first 10 rows of data, and then repeat the import using a `while` loop, 10 rows at a time. Continue until all data has been retrieved, which occurs when `curs.Data` is `'No Data'`.

```
% Initialize RowLimit (fetchsize)
fetchsize = 10
% Check for more data. Retrieve and display all data.
while ~strcmp(curs.Data, 'No Data')
 curs=fetch(curs,fetchsize);
 curs.Data(:)
end
```

When processing terminates, MATLAB returns

```
ans =
    'No Data'
```

Example 4 — Import Numeric Data

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

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

MATLAB retrieves the data into a numeric matrix.

```
ans =
    39
    17
    13
```

Example 5 — Import BOOLEAN Data

Import data that includes a BOOLEAN field. Use setdbprefs to specify the format for the retrieved data as cellarray.

```
conn = database('SampleDB', '', '');
curs=exec(conn, 'select ProductName, ...
Discontinued fromProducts');
setdbprefs('DataReturnFormat','cellarray')
curs=fetch(curs,5);
A=curs.Data
A =
    'Chai'                [0]
    'Chang'               [0]
    'Aniseed Syrup'       [0]
        [1x28 char]       [0]
        [1x22 char]       [1]
```

View the class of the second column in A.

```
class(A{1,2}
ans =
logical
```

See Also

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

“Retrieving BINARY or OTHER Java SQL Data Types” on page 3-25

database

Purpose

Connect to database

GUI Alternatives

As an alternative to the `database` function, you can connect to databases using Visual Query Builder. Run `querybuilder` to access it and use the **Help** menu for more information.

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" on page 1-13. `username` and `password` are the username and 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. After connecting, use `exec` to retrieve data.

`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. The JDBC driver is sometimes referred to as the class that implements the Java SQL driver for your database. `databaseurl` is the JDBC URL object of the form `jdbc:`*`subprotocol`*`:`*`subname`*. The *`subprotocol`* is a database type, such as `oracle`. The *`subname`* might contain other information used by `driver`, such as the location of the database and/or a port number. The *`subname`* might take the form `//`*`hostname`*`:`*`port`*`/`*`databasename`*. Find the correct `driver` name and `databaseurl` format in the driver manufacturer's documentation. Some sample `databaseurl` strings are listed in "Example 3 — Establish JDBC Connection" on page 5-30.

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

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

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

You can have multiple database connections open at one time.

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

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

Examples

Example 1 — Establish ODBC Connection

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

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

Example 2 — Establish ODBC Connection Without Username and Password

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

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

Example 3 — Establish JDBC Connection

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

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

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

Database	JDBC Driver and Database URL Examples
Informix	JDBC driver: `com.informix.jdbc.IfxDriver` Database URL: `jdbc:informix-sqli://161.144.202.206:3000:INFORMIXSERVER=stars`
MySQL	JDBC driver: `twz1.jdbc.mysql.jdbcMysqlDriver` Database URL: `jdbc:z1MySQL://natasha:3306/metrics` JDBC driver: `com.mysql.jdbc.Driver` Database URL: `jdbc:mysql://devmetrics.mrkps.com/testing`
Oracle oci7 drivers	JDBC driver: `oracle.jdbc.driver.OracleDriver` Database URL: `jdbc:oracle:oci7:@rex`
Oracle oci8 drivers	JDBC driver: `oracle.jdbc.driver.OracleDriver` Database URL: `jdbc:oracle:oci8:@111.222.333.44:1521:` Database URL: `jdbc:oracle:oci8:@frug`
Oracle thin drivers	JDBC driver: `oracle.jdbc.driver.OracleDriver` Database URL: `jdbc:oracle:thin:@144.212.123.24:1822:`

Database	JDBC Driver and Database URL Examples
Oracle 10 connections with JDBC (thin drivers)	JDBC driver: `oracle.jdbc.driver.OracleDriver` Database URL: `jdbc:oracle:thin:` (do not specify the target name and port)
PostgreSQL	JDBC driver: `org.postgresql.Driver` Database URL: `jdbc:postgresql://masd/MOSE`
PostgreSQL with SSL connection	JDBC driver: `org.postgresql.Driver` Database URL: `jdbc:postgresql:servername:dbname:ssl=true&sslfactory=org.postgresql.ssl.NonValidatingFactory&` (the trailing & is required)
Microsoft SQL Server	JDBC driver: `com.microsoft.jdbc.sqlserver.SQLServerDriver` Database URL: `jdbc:microsoft:sqlserver://127.0.0.1:1403` JDBC driver: `com.inet.tds.TdsDriver` Database URL: `jdbc:inetdae:sqlgckprod:1433?database=gck`
Sybase SQL Server and SQL Anywhere	JDBC driver: `com.sybase.jdbc.SybDriver` Database URL: `jdbc:sybase:Tds:yourhostname:yourportnumber/`

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

For Microsoft SQL Server 2000, you may also need to pass the database name, username, and password via the URL. For example,

```
conn = database('pubs','sa','sec',
'com.microsoft.jdbc.sqlserver.SQLServerDriver',
'jdbc:microsoft:sqlserver://127.0.0.1:1403;
database=pubs;user=sa;password=sec')
```

See Also

`close`, `dmd`, `exec`, `fastinsert`, `get`, `getdatasources`, `isconnection`, `isreadonly`, `logintimeout`, `ping`, `supports`, `update`

Purpose

Execute SQL statement and import data into MATLAB

Syntax

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

Description

`results = fetch(conn, sqlquery)` executes the valid SQL statement `sqlquery`, and imports data given the open connection object `conn` (created using `database`). `results` is a cell array, structure, or numeric matrix, based on specifications made using `setdbprefs`. For more information about valid SQL statements, see `exec`.

`results = fetch(conn, sqlquery, RowInc)` executes the valid SQL statement `sqlquery`, and imports rows of data `RowInc` at a time, given the open connection object `conn` (created using `database`). For more information about valid SQL statements, see `exec`. Data is stored in MATLAB in a cell array, structure, or numeric matrix, based on specifications made using `setdbprefs`. If you import large amounts of data that cause memory or speed problems, use `RowInc` to address them. Regardless of the value for `RowInc`, all data is returned when you run `fetch`; `RowInc` is used internally for speed and memory management purposes.

Remarks

Note This page documents `fetch` for a database object. For more information about the relationship with `cursor.fetch`, see `fetch`.

Do not count on the order of records in your database as being constant, but rather always use the values in column names to identify records. You can use the SQL `ORDER BY` command in your `sqlquery` statement to sort the data.

When a fetched field contains `BOOLEAN` data, it is represented as a `logical` data type in MATLAB.

When the results contain `BINARY` or `OTHER` data types, you might need to understand the content and process it before using it in MATLAB. See

"Retrieving BINARY or OTHER Java SQL Data Types" on page 3-25 for a specific example about processing bitmap image data.

Examples

Example 1 — Import Data

Import the `country` column of data from the `customers` table in the `SampleDB` database:

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

results =

    'Germany'
    'Mexico'
    'Mexico'
    'UK'
    'Sweden'

    ...

    'Finland'
    'Brazil'
    'USA'
    'Finland'
    'Poland'
```

Run

```
size(results)
```

and MATLAB returns

```
ans =

    91     1
```

indicating that 91 rows of data were imported.

Example 2 — Import Data Using RowInc

This is identical to Example 1, Import Data, however, it also uses the `RowInc` argument to avert potential memory and speed problems:

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

results =

    'Germany'
    'Mexico'
    'Mexico'
    'UK'
    'Sweden'

    ...

    'Finland'
    'Brazil'
    'USA'
    'Finland'
    'Poland'
```

Run

```
size(results)
```

and MATLAB returns

```
ans =

     91     1
```

indicating that 91 rows of data were imported.

The results are identical to those in Example 1. `RowInc` does not affect the number of rows of data retrieved, but rather is used internally by fetch. If there were speed or memory usage problems when running Example 1, using RowInc might help to resolve them.

Example 3 — Import Two Columns of Data and Get Information

Import the ProductName and Discontinued columns from the SampleDB database:

```
conn = database('SampleDB', '', '');
setdbprefs('DataReturnFormat','cellarray')
results=fetch(conn, 'select ProductName, Discontinued from Products');
```

Run

```
size(results)
```

to see that there are 77 rows and 2 columns of data:

```
ans =

    77   2
```

To see the results for the first row of data, run

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

and MATLAB returns

```
ans =

    'Chai'    [0]
```

You can retrieve some information about the data. For example, run

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

and MATLAB returns

```
ans =
logical
```

If you want more information about the data, such as column names and attributes of the data such as the size of the field, instead use `cursor.fetch` and the functions that use the resulting cursor object, such as `columnnames` and `attr`.

See Also

`cursor.fetch`, `database`, `exec`, `fetch`, `logical`,

"Retrieving BINARY or OTHER Java SQL Data Types" on page 3-25

Purpose

Construct database metadata object

Syntax

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

Description

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

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

```
help dmd/Contents
```

Examples

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

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

See Also

`columns`, `database`, `get`, `supports`, `tables`

Purpose

Construct database driver object

Syntax

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

Description

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

Examples

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

See Also

`get`, `isdriver`, `isjdbc`, `isurl`, `register`

drivermanager

Purpose

Construct database drivermanager object

Syntax

```
dm = drivermanager
```

Description

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

Examples

`dm = drivermanager` creates the database drivermanager object `dm`.

`get(dm)` returns the properties of the drivermanager object `dm`.

See Also

`get`, `register`, `set`

Purpose

Execute SQL statement and open cursor

GUI Alternatives

As an alternative to the `exec` function, you can query databases using Visual Query Builder. Run `querybuilder` to access it and use the **Help** menu for more information.

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 be a stored procedure for that database connection, of the form `{call sp_name (parm1,parm2,...)}`.

Remarks

- After opening a cursor, use `fetch` to import data from the cursor. Use `resultset`, `rsmd`, and `statement` to get properties of the cursor.
- Use `querytimeout` to determine the maximum amount of time for which `exec` will try to complete the SQL statement.
- You can have multiple cursors open at one time.
- A cursor stays open until you close it using the `close` function. Always close a cursor after you finish using it.
- Perform database administrative tasks, such as creating tables, using your database system application. Database Toolbox is not intended to be used as a tool for database administration.
- Unless specifically noted in this reference page, all valid SQL statements, such as nested queries, are supported by the `exec` function.
- Do not count on the order of records in your database as being constant, but rather always use the values in column names to identify records. Use the SQL `ORDER BY` command to perform sorting.
- If you attempt to modify database tables from Database Toolbox, be sure that you (or another user for a shared database) do not have the

database open for editing (design mode in Microsoft Access). If the database is open for editing and you try to modify it, you will receive the following error in MATLAB.

```
[Vendor][ODBC Driver] The database engine could not lock
table 'TableName' because it is already in use by
another person or process.
```

- For Microsoft Excel, tables in `sqlquery` are Excel sheets. By default, some sheet names include `$`. To select data from a sheet with this name format, the SQL statement should be of this form: `select * from "Sheet1$"` (or `'Sheet1$'`).
- For the Microsoft SQL Server database management system, you might experience problems with text field formats. One workaround is to convert fields of the formats `NVARCHAR`, `TEXT`, `NTEXT`, and `VARCHAR` to `CHAR` on the database side. Another possible workaround is to convert data to `VARCHAR` as part of `sqlquery`. As an example, use a `sqlquery` of the form `'select convert(varchar(20), field1) from table1'`
- The PostgreSQL database management system supports multidimensional fields, but SQL `select` statements fail when getting these fields unless an index is specified.
- Some databases require that you include the # symbol before and after a date in a query. Some databases use a different symbol, while most require none. For example:

```
curs = exec(conn,'select * from mydb where mydate > #03/05/2005#')
```

Examples

Example 1 — Select All Data from Database Table

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

```
curs = exec(conn, 'select * from customers')
curs =
    Attributes: []
          Data: 0
DatabaseObject: [1x1 database]
      RowLimit: 0
      SQLQuery: 'select * from customers'
       Message: []
          Type: 'Database Cursor Object'
     ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
        Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
     Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
         Fetch: 0
```

Example 2 — Select One Column of Data from Database Table

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

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

Example 3 — Use Variable in a Query

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

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

MATLAB prompts

```
Enter your country:
```

The user responds

```
Mexico
```

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

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

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

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

The select statement is created by using square brackets to concatenate the two strings `select * from customers where country =` and `'UserCountry'`. The pair of four quotation marks are needed to create the pair of single quotation marks that appear in the SQL statement around `UserCountry`—the outer two marks delineate the next string to be concatenated, and inside them, two marks are required to denote a quotation mark inside a string.

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

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

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

To commit the data, type:

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

Example 5 — Run Stored Procedure

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

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

You can run a stored procedure with input parameters. For example:

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

Example 6 — Change Catalog

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

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

See Also

`close`, `cursor.fetch`, `database`, `database.fetch`, `fastinsert`, `fetch`, `procedures`, `querybuilder`, `querytimeout`, `resultset`, `rsmd`, `set`, `update`

"Data Retrieval Restrictions" on page 1-8

exportedkeys

Purpose

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 the Java Web site for the getExporetedKeys property.

See Also

crossreference, dmd, get, importedkeys, primarykeys

Purpose

Add MATLAB data to database table

GUI Alternatives

As an alternative to the `fastinsert` function, you can export data using Visual Query Builder, with the **Data operation** set to **Insert**. Note that VQB actually uses the `insert` function instead of `fastinsert`.

Syntax

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

Description

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

The status of the `AutoCommit` flag determines if `fastinsert` 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`.

Note The `rollback` function does not roll back data in a MySQL database.

Remarks

The `fastinsert` function replaces the `insert` function. It improves upon `insert` by offering better performance and supporting more object types. If `fastinsert` does not work as expected, try `insert` instead, especially if you used `insert` successfully in the past. The `insert` function has the same syntax as `fastinsert`. Note that VQB uses `insert` instead of `fastinsert`.

Do not count on the order of records in your database as being constant, but rather always use the values in column names to identify records.

If you get an error when you use `fastinsert`, it might be because the table is open in design mode in Access (edit mode for other databases). Close the table in the database and repeat the `fastinsert` function. For example, the error might be

```
[Vendor][ODBC Product Driver] The database engine could
not lock table 'TableName' because it is already in use
by another person or process.
```

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.

```
fastinsert(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 multiple 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.

```
fastinsert(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 from the products table. Because the data contains numeric and character data, import the data into a cell array.

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

Assign the first column of data to the variable prod_name.

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

Assign the sixth column of data to the variable price.

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

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

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

Export the prod_name, price, and new_price data to the Sale table. Because prod_name is a character array and price is numeric, export the data as a cell array, which supports mixed data types. The variable new_price is a numeric matrix because it was the result of the discount calculation. You must convert new_price to a cell array. To convert the columns of data in new_price to a cell array, type

```
new_price = num2cell(new_price);
```

Create an array, exdata, that contains the three columns of data to be exported. Put the prod_name data in column 1, price in column 2, and new_price in column 3.

```
exdata(:,1) = prod_name(:,1);
exdata(:,2) = price;
exdata(:,3) = new_price;
```

Assign the column names to a string array, colnames.

```
colnames={'product_name', 'price', 'sale_price'};
```

Export the data to the Sale table.

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

All rows of data are inserted into the Sale table.

Example 4 — Insert Numeric Data

Export the tax_rate data into the Tax table, where tax_rate is a numeric matrix consisting of two columns:

```
fastinsert(conn, 'Tax', {'rate','max_value'}, tax_rate)
```

When exporting, you do not need to define the type of data you are exporting. The format in setdbprefs does not apply when exporting data from MATLAB.

Example 5 — Insert Followed by commit

This example demonstrates the use of the SQL commit function following an insert. The AutoCommit flag is off.

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

```
fastinsert(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');
```

Example 6 — Insert BOOLEAN Data

Insert `BOOLEAN` data (the `logical` data type in MATLAB) from MATLAB to a database.

```
conn = database('SampleDB', '', '');
P.ProductName{1}='Chocolate Truffles';
P.Discontinued{1}=logical(0);
fastinsert(conn,'Products',...
 {'ProductName';'Discontinued'}, P)
```

View the new record in the database to verify that value in the `Discontinued` field is `BOOLEAN`. For some databases, the MATLAB logical `0` is shown as a `BOOLEAN` `false` (or `No` or a cleared check box).

See Also

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

Purpose

`cursor.fetch` or `database.fetch`

About fetch, cursor.fetch, and database.fetch

There are two `fetch` functions in Database Toolbox, `cursor.fetch` and `database.fetch`. You use the syntax `fetch` along with the appropriate object argument—you do *not* explicitly specify `cursor.fetch` or `database.fetch`. When Database Toolbox runs `fetch`, it uses `cursor.fetch` or `database.fetch`, depending on the object you provided as an argument to `fetch`.

For example, Database Toolbox uses `cursor.fetch` when you run

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

because you supplied a cursor object, `curs`, as the argument to `fetch`.

Alternatively, Database Toolbox uses `database.fetch` when you run

```
conn=database(...)
fetch(conn, sqlquery)
```

because you supplied a database object, `conn`, as the argument to `fetch`.

In this example, the results are effectively identical—`database.fetch` is a convenient, but limited alternative that allows you to accomplish with one statement results similar to running two statements, `exec` and `cursor.fetch`. `database.fetch` runs `exec`, returns results to the cursor object, runs `cursor.fetch`, returns results, and closes the cursor object.

However, `cursor.fetch` returns a cursor object on which you can perform many other Database Toolbox functions, such as `get` and `rows`. For this reason, `cursor.fetch` is the recommended usage in most situations. If your intention is to simply import data into MATLAB without the need for meta information about the data, you can use `database.fetch` instead of `cursor.fetch`, knowing the limitations of the results.

Throughout the documentation, references to `fetch` imply `cursor.fetch` unless explicitly stated otherwise.

The only instances in which you can specify `database.fetch` or `cursor.fetch` explicitly are when running `help` or `doc`. For example, help `fetch` displays help for `cursor.fetch` and provides a link to help for database/fetch, which is an alternative way of displaying `database.fetch`, as an overloaded function. To get help directly for `database.fetch`, run `help database.fetch`. Similarly, to view the reference pages for either version of `fetch` directly, run `doc database.fetch` or `doc cursor.fetch`.

See Also

`cursor.fetch`, `database`, `database.fetch`, `exec`

fetchmulti

Purpose

Import data into MATLAB from multiple resultsets

Syntax

```
curs = fetchmulti(curs)
```

Description

`curs = fetchmulti(curs)` imports data from the open SQL cursor object `curs` (created using `exec`, into the object `curs` and supports the case when the open SQL cursor object contains multiple resultsets. Multiple resultsets are retrieved via `exec` with a `sqlquery` statement that runs a stored procedure containing two select statements. `cursmulti.Data` contains the data from each resultset associated with `cursmulti.Statement`. `curmulti.Data` is a cell array consisting of either cell arrays, structures, or numeric matrices, as specified via `setdbprefs`; the data type is the same for all resultsets.

Examples

This example shows how to use `exec` to run a stored procedure that includes multiple `select` statements and `fetchmulti` to retrieve the resulting multiple resultsets.

```
conn = database(...)
setdbprefs('DataReturnFormat','cellarray')
curs = exec(conn, '{call sp_1}');
curs = fetchmulti(curs)
```

MATLAB returns

```
Attributes: []
              Data: {{10x1 cell} {12x4 cell}}
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: '{call sp_1}'
           Message: []
              Type: 'Database Cursor Object'
         ResultSet: [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
          [1x1 sun.jdbc.odbc.JdbcOdbcResultSet]
            Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
         Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
           [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
         Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

See Also

`cursor.fetch`, `database`, `exec`, `setdbprefs`

Purpose

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” on page 5-59, created using `database`
- “Cursor Object” on page 5-60, created using `exec` or `fetch` (`cursor.fetch`)
- “Driver Object” on page 5-61, created using `driver`
- “Database Metadata Object” on page 5-61, created using `dmd`
- “Drivermanager Object” on page 5-62, created using `drivermanager`
- “Resultset Object” on page 5-62, created using `resultset`
- “Resultset Metadata Object” on page 5-63, created using `rsmd`

If you are calling these objects from your own Java-based applications, see the Java Web site 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'`	Name of the catalog in the data source, for example, `'Nwind'`. Note that functions using the `cata` argument, such as `columns`, accept only a single catalog, so you might need to extract a single catalog name from `'Catalog'`.
`'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. This field is always empty—for cursor attributes, use the `attr` function.
`'Data'`	Data in the cursor object data element (the query results)
`'DatabaseObject'`	Information about the database object
`'RowLimit'`	Maximum number of rows to be returned by `fetch`, as specified by `set`
`'SQLQuery'`	SQL statement for the cursor, as specified by `exec`
`'Message'`	Error message returned from `exec` or `fetch`
`'Type'`	Object type, specifically `Database Cursor Object`
`'ResultSet'`	Resultset object identifier
`'Cursor'`	Cursor object identifier
`'Statement'`	Statement object identifier
`'Fetch'`	`0` for cursor created using `exec`; `fetchTheData` for cursor created using `fetch`

Driver Object

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

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

Database Metadata Object

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

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

Drivermanager Object

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

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

Resultset Object

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

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

Resultset Metadata Object

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

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

The empty strings for `CatalogName` and `TableName` indicate that the database does not return these values.

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

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

Examples

Example 1 — Get Connection Property, Data Source Name

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

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

Example 2 — Get Connection Property, AutoCommit Flag Status

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

```
get(conn, 'AutoCommit')

ans =
 on
```

Example 3 — Display Data in Cursor

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

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

or by typing

```
curs.Data
```

MATLAB returns

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

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

Example 4 — Get Database Metadata Object Properties

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

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

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

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

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

```
v.Catalogs
```

MATLAB returns the catalog names

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

See Also

`columns`, `cursor.fetch`, `database`, `dmd`, `driver`, `drivermanager`, `exec`, `getdatasources`, `resultset`, `rows`, `rsmd`, `set`

getdatasources

Purpose

Names of valid ODBC and JDBC data sources on system

Syntax

```
d = getdatasources
```

Description

`d = getdatasources` returns the names of valid ODBC and JDBC data sources on the system as a cell array of strings. The function gets the names of ODBC data sources from the `ODBC.INI` file located in the directory returned by running

```
myODBCdir = getenv('WINDIR')
```

If `d` is empty, the `ODBC.INI` file is valid but no data sources have been defined. If `d` equals `-1`, the `ODBC.INI` file could not be opened. The function also gets the names of data sources in the system registry that are not in the `ODBC.INI` file.

If you do not have write access to `myODBCdir`, the results of `getdatasources` might not include data sources you recently added. In that event, specify a temporary, writable, output directory via the preference `TempDirForRegistryOutput`—for more information, see `setdbprefs`.

`getdatasources` gets the names of JDBC data sources from the file defined using `setdbprefs` or the Define JDBC Data Sources dialog box (`confds`).

Examples

Type

```
d = getdatasources
```

MATLAB returns the three valid databases on the system:

```
d =
    'MS Access Database'    'SampleDB'    'dbtoolboxdemo'
```

See Also

`database`, `get`, `setdbprefs`

Purpose

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`, which 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 the Java Web site for the `getImportedKeys` property.

See Also

`crossreference`, `dmd`, `exportedkeys`, `get`, `primarykeys`

indexinfo

Purpose

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 the Java Web site for a description of the `getIndexInfo` property.

See Also

`dmd`, `get`, `tables`

Purpose

Add MATLAB data to database table (deprecated; use `fastinsert` instead)

Syntax

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

Description

`insert(conn, 'tab', colnames, exdata)` The `insert` function was replaced by `fastinsert`, which offers improved performance and supports more data types. Use `insert` if `fastinsert` does not work as you expected, especially if you used `insert` successfully in the past.

The `insert` function uses the same syntax as `fastinsert`; for details, see `fastinsert`.

Note that VQB uses `insert` instead of `fastinsert`.

See Also

`commit`, `fastinsert`, `querybuilder`, `rollback`

isconnection

Purpose Detect whether 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`

isdriver

Purpose

Detect whether driver is valid JDBC driver object

Syntax

```
a = isdriver(d)
```

Description

`a = isdriver(d)` returns `1` if `d` is a valid JDBC driver object, 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`

Purpose

Detect whether driver is JDBC compliant

Syntax

```
a = isjdbc(d)
```

Description

`a = isjdbc(d)` returns `1` if the driver object `d` is JDBC compliant, 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`

isnullcolumn

Purpose

Detect whether 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 whether 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 `fastinsert`, `insert`, or `update` functions for this database.

See Also

`database`, `isconnection`

Purpose

Detect whether database URL is valid

Syntax

```
a = isurl('s', d)
```

Description

`a = isurl('s', d)` returns `1` if the database URL `s`, for the driver object `d`, is valid, or returns `0` otherwise. The URL `s` is of the form `jdbc:odbc:`*`name`* or *`name`*, and `d` is the driver object created using `driver`.

Examples

Type

```
a = isurl('jdbc:odbc:thin:@144.212.123.24:1822:', d)
```

and MATLAB returns

```
a =
     1
```

indicating that the database URL, `jdbc:odbc:thin:@144.212.123.24:1822:`, is valid for driver object d.

See Also

`driver`, `get`, `isdriver`, `isjdbc`

logintimeout

Purpose

Set or get time allowed to establish database connection

Syntax

```
timeout = logintimeout('driver', time)
timeout = logintimeout(time)
timeout = logintimeout('driver')
timeout = logintimeout
```

Description

`timeout = logintimeout('driver', time)` sets the amount of time, in seconds, 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 0 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 0 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 freeze.

Note On the Macintosh platform, `logintimeout` is not supported.

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 `5` 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 `5` seconds. Type

```
  timeout = logintimeout('oracle.jdbc.driver.OracleDriver',5)
```

MATLAB returns

```
timeout =
     5
```

Verify the timeout value for the JDBC driver. Type

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

MATLAB returns

```
ans =
     5
```

See Also `database`, `get`, `set`

Purpose

Map resultset column name to resultset column index

Syntax

```
x = namecolumn(rset, n)
```

Description

`x = namecolumn(rset, n)` maps a resultset column name `n`, to its resultset column index, 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

Status information about database connection

Syntax

```
ping(conn)
```

Description

`ping(conn)` returns the status information about the database connection, `conn`. If the connection is open, `ping` returns status information and otherwise it returns an error message.

Examples

Example 1 — Get Status Information About ODBC Connection

Type

```
ping(conn)
```

where `conn` is a valid ODBC connection. MATLAB returns

```
ans =
      DatabaseProductName: 'ACCESS'
   DatabaseProductVersion: '03.50.0000'
           JDBCDriverName: 'JDBC-ODBC Bridge (odbcjt32.dll)'
        JDBCDriverVersion: '1.1001 (04.00.4202)'
   MaxDatabaseConnections: 64
          CurrentUserName: 'admin'
              DatabaseURL: 'jdbc:odbc:SampleDB'
   AutoCommitTransactions: 'True'
```

Example 2 — Get Status Information About JDBC Connection

Type

```
ping(conn)
```

where `conn` is a valid JDBC connection.

MATLAB returns

```
ans =
    DatabaseProductName: 'Oracle'
 DatabaseProductVersion: [1x166 char]
         JDBCDriverName: 'Oracle JDBC driver'
      JDBCDriverVersion: '7.3.4.0.2'
 MaxDatabaseConnections: 0
        CurrentUserName: 'scott'
            DatabaseURL: 'jdbc:oracle:thin:@144.212.123.24:
                         1822:orcl'AutoCommitTransactions:'True'
```

Example 3 — Unsuccessful Request for Information About Connection

Type

```
ping(conn)
```

where the database connection `conn` has been terminated or was not successful. MATLAB returns

```
Cannot Ping the Database Connection
```

See Also

`database`, `dmd`, `get`, `isconnection`, `set`, `supports`

primarykeys

Purpose

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`

procedurecolumns

Purpose

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 the Java Web site for the `getProcedureColumns` property.

See Also

`dmd`, `get`, `procedures`

Purpose

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`

querybuilder

Purpose

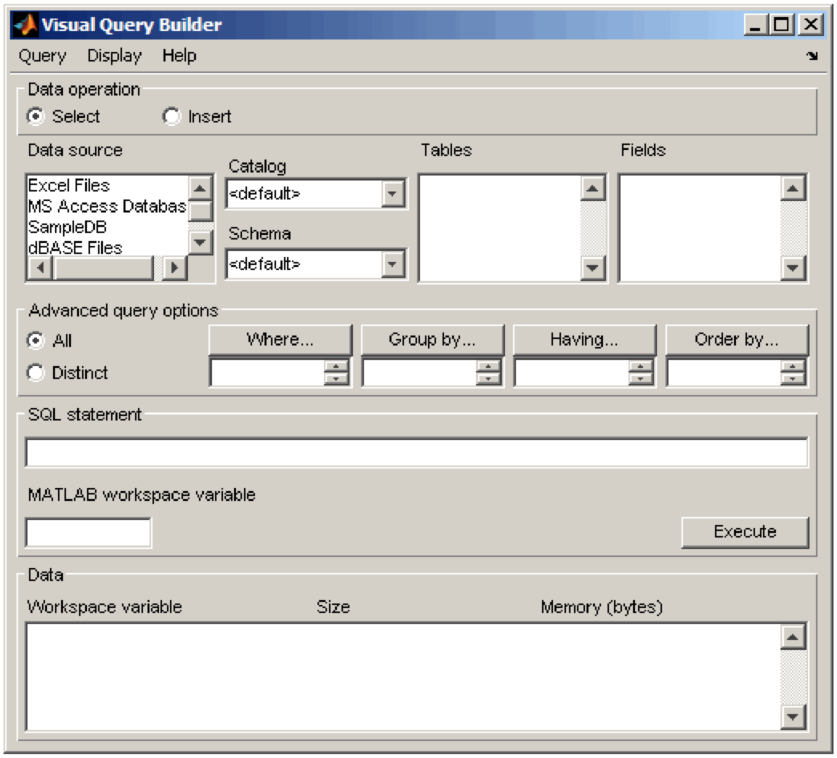
Start SQL query builder GUI to import and export data

Syntax

```
querybuilder
```

Description

`querybuilder` starts Visual Query Builder (VQB), an easy-to-use interface for building and running SQL queries to exchange data with databases.



Examples

For examples of and more information about using Visual Query Builder, use VQB **Help** menu or see Chapter 2, “Visual Query Builder”. You can also get help in any of Visual Query Builder dialog boxes by clicking the **Help** button in the dialog box.

Purpose

Time allowed for 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`

register

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 use `get` to view properties of the driver before connecting. The `register` function also allows you to use `drivermanager` with `set` and `get` for 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`, `set`, `unregister`

Purpose

Construct resultset object

Syntax

```
rset = resultset(curs)
```

Description

`rset = 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`, `cursor.fetch`, `exec`, `get`, `isnullcolumn`, `namecolumn`, `rsmd`

rollback

Purpose

Undo database changes

Syntax

```
rollback(conn)
```

Description

`rollback(conn)` reverses changes made via `fastinsert`, `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

```
fastinsert(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 `fastinsert`.

See Also

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

Purpose

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`, `cursor.fetch`, `get`, `rsmd`

Purpose

Construct resultset metadata object

Syntax

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

Description

`rsmeta = rsmd(rset)` creates a resultset metadata object `rsmeta`, for the resultset object `rset`, 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`

Purpose

Call stored procedure with input and output parameters

Syntax

```
results = runstoredprocedure(conn, sp_name, parms_in,
   types_out)
```

Description

`results = runstoredprocedure(conn, sp_name, parms_in, types_out)` calls a stored procedure with specified input parameters and returns output parameters, where `conn` is the database connection handle, `sp_name` is the stored procedure to be run, `parms_in` is a cell array containing the stored procedure's input parameters, and `types_out` is the list of data types of the output parameters. Use `runstoredprocedure` to return the value of a variable to a MATLAB variable, which you cannot do when you run a stored procedure via `exec`. Running a stored procedure via `exec` returns resultsets but cannot return output parameters.

Examples

These examples illustrate the use of `runstoredprocedure` relative to running a stored procedure via `exec`.

```
x = runstoredprocedures(c,'myprocnoparams')
```

runs a stored procedure that has no input or output parameters. This could also be accomplished with an `exec` statement of the form `exec(c, 'myprocnoparams')`.

```
x = runstoredprocedure(c,'myprocinonly',{2500,'Jones'})
```

runs a stored procedure given the input parameters `2500` and `'Jones'`. It returns no output parameters. This could also be accomplished with an `exec` statement of the form `exec(c, '{call myprocinonly (2500,Jone)}'`

```
x = runstoredprocedure(c,'myproc',{2500,'Jones'},{java.sql.Types.NUMERIC})
```

runs the stored procedure `myproc` given the input parameters `2500` and `'Jones'`. It returns an output parameter of type `java.sql.Types.NUMERIC`, which could be any numeric Java data type. The output parameter `x` is, the value of a database variable `n`, created by running the stored procedure `myproc`, given the input values `2500` and `'Jones'`. For example, `myproc` computes `n`, the number of days when `Jones` is `2500`, with the result being `14`, and returns the value of `n` to `x`, so `x` is `14` in MATLAB.

This cannot be accomplished via an `exec` statement, because `exec` does not support stored procedures that return output parameters; this the primary advantage of `runstoredprocedure`.

See Also `cursor.fetch`, `exec`

Purpose

Set properties for database, cursor, or drivermanager object

Syntax

```
set(object, 'property', value)
set(object)
```

Description

`set(object, 'property', value)` sets the value of *`property`* to `value` for the specified `object`.

`set(object)` displays all properties for `object`.

Allowable values you can set for `object` are

- “Database Connection Object” on page 5-104, created using `database`
- “Cursor Object” on page 5-105, created using `exec` or `fetch` (`cursor.fetch`)
- “Drivermanager Object” on page 5-105, 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 a fastinsert, 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 a fastinsert, insert, or update function. In this case, after you run fastinsert, insert, or update, you can use rollback to reverse it. When you are sure the data is correct, follow a fastinsert, 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, fastinsert, 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:@144.212.123.24:1822:');
curs=exec(conn, 'select * from EMP');
set(curs, 'RowLimit', 5)
curs=fetch(curs)
curs =
     Attributes: []
           Data: {5x8 cell}
 DatabaseObject: [1x1 database]
       RowLimit: 5
       SQLQuery: 'select * from EMP'
        Message: []
           Type: 'Database Cursor Object'
      ResultSet: [1x1 oracle.jdbc.driver.OracleResultSet]
         Cursor: [1x1 com.mathworks.toolbox.database.sqlExec]
      Statement: [1x1 oracle.jdbc.driver.OracleStatement]
          Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

As seen above, the `RowLimit` property of `curs` is now `5` and the `Data` property is `5x8 cell`, meaning five rows of data were returned.

For the database in this example, the `RowLimit` acts as the maximum number of rows you can retrieve. Therefore, if you run the `fetch` function again, no data is returned.

Example 2 — Set AutoCommit Flag to On for Connection

This example shows a database `update` when the `AutoCommit` flag is `on`. First determine the status of the `AutoCommit` flag for the database connection `conn`.

```
get(conn, 'AutoCommit')

ans =
off
```

The flag is `off`.

Set the flag status to `on` and verify it.

```
set(conn, 'AutoCommit', 'on');
get(conn, 'AutoCommit')

ans =
on
```

Insert data, cell array `exdata`, into the column names `colnames`, of the `Growth` table.

```
fastinsert(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 `fastinsert` 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.

```
fastinsert(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

`cursor.fetch`, `database`, `drivermanager`, `exec`, `fastinsert`, `get`, `insert`, `logintimeout`, `ping`, `update`

setdbprefs

Purpose

Set preferences for retrieval format, errors, `NULL`s, and more

GUI Alternatives

As an alternative to the `setdbprefs` function, you can select **Preferences** from the Visual Query Builder **Query** menu and use the Preferences dialog box.

Syntax

```
setdbprefs
s = setdbprefs
setdbprefs('property')
setdbprefs('property', 'value')
setdbprefs({'property1'; ...},{'value1'; ...})
setdbprefs(s)
```

Description

`setdbprefs` returns the current values for database preferences.

`s = setdbprefs` returns the current values for database preferences to the structure `s`. If you repeatedly use certain sets of preferences, save `s` to a MAT-file for later reuse.

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

`setdbprefs('`*`property`*`', '`*`value`*`')` sets the specified preference `property` to `value` for the current MATLAB session. Include the statement in a MATLAB `startup` file to set preferences automatically for the session when MATLAB starts.

`setdbprefs({'`*`property1`*`'; ...},{'`*`value1`*`'; ...})` for the properties starting with `property1`, sets the preference values starting with `value1`, for the current session.

`setdbprefs(s)` sets preferences specified in the structure `s` to the values specified for `s`.

Allowable properties are listed in the following tables:

- DataReturnFormat and ErrorHandling Properties and Values for setdbprefs on page 5-112

- Null Data Handling Properties and Values for setdbprefs on page 5-113
- Other Properties and Values for setdbprefs (Not Accessible via Query > Preferences) on page 5-114

DataReturnFormat and ErrorHandling Properties and Values for setdbprefs

Property	Allowable Values	Description
`'DataReturnFormat'`	`'cellarray'` (default), `'numeric'`, or `'structure'`	Format for data imported into MATLAB. Select a value based on the type of data you are importing, memory considerations, and your preferred method of working with retrieved data. Set the value before using `fetch`.
	`'cellarray'` (default)	Imports data into MATLAB cell arrays. Use for nonnumeric data types. Requires substantial system memory when retrieving large amounts of data. Has slower performance than `numeric` format. To address memory problems, use the `RowLimit` option with `fetch`. For more information about cell arrays, see "Working with Cell Arrays in MATLAB" on page 3-40.
	`'numeric'`	Imports data into a MATLAB matrix of doubles. Nonnumeric data types are considered to be `NULL` numbers and are shown as specified for the `NullNumberRead` property. Uses less system memory and offers better performance than the `cellarray` format. Use only when data to be retrieved is in numeric format, or when the nonnumeric data retrieved is not relevant.
	`'structure'`	Imports data as a MATLAB structure. Can use for all data types. Makes it easy to work with returned columns. Requires substantial system memory when retrieving large amounts of data. Has slower performance than `numeric` format. To address memory problems, use the `RowLimit` option with `fetch`. For more information on using

Property	Allowable Values	Description
		structures, see "Data Types" in the MATLAB Programming documentation.
'ErrorHandling'	'store' (default), 'report', or 'empty'	Behavior for handling errors when importing data. Set the value before running exec.
	'store' (default)	Any errors from running database are stored in the Message field of the returned connection object. Any errors from running exec are stored in the Message field of the returned cursor object.
	'report'	Any errors from running database or exec display immediately in the Command Window.
	'empty'	Any errors from running database are stored in the Message field of the returned connection object. Any errors from running exec are stored in the Message field of the returned cursor object. Objects that cannot be created are returned as empty handles, [ ].

Null Data Handling Properties and Values for setdbprefs

Property	Allowable Values	Description
'NullNumberRead'	User-specified, for example, '0'	How NULL numbers in a database are represented when imported into MATLAB. NaN is the default value. Cannot specify a string value, such as 'NULL', if 'DataReturnFormat' is set to 'numeric'. Set the value before using fetch.

Property	Allowable Values	Description
'NullNumberWrite'	User-specified, for example, 'NaN'	Any numbers in the specified format, for example, NaN are represented as NULL when exported to a database. NaN is the default value.
'NullStringRead'	User-specified, for example, 'null'	How NULL strings in a database are represented when imported into MATLAB. NaN is the default value. Set the value before using fetch.
'NullStringWrite'	User-specified, for example, 'NULL'	Any strings in the specified format, for example, NaN, are represented as NULL when exported to a database. NaN is the default value.

Other Properties and Values for setdbprefs (Not Accessible via Query > Preferences)

Property	Allowable Values	Description
'JDBCDataSourceFile'	User-specified, for example, 'D:/file.mat'	Full pathname to MAT-file containing JDBC data sources defined using Visual Query Builder. For more information, see "Define a JDBC Data Source in Visual Query Builder" on page 1-23. The graphical interface for setting this preference is in VQB: select **Query > Define JDBC Data Source**, and then click **Use Existing File**. If VQB is open, close it and reopen it to use the data source specified via setdbprefs.

Property	Allowable Values	Description
'UseRegistryForSources'	'yes' or 'no'	When set to yes, the default, instructs Visual Query Builder to search the Windows registry to find any ODBC data sources not uncovered using the system ODBC.INI.
TempDirForRegistryOutput	User-specified, for example, 'D:/work'	Full pathname to directory where Visual Query Builder temporarily writes ODBC registry settings when running getdatasources. Use when you add data sources and do not have write access for the MATLAB current directory. Without write access, getdatasources does not always return the data sources you added. You must have write access to the directory you specify.

Remarks

When you run `clear all`, the `setdbprefs` values are cleared and return to default values. It is a good practice to set or verify the `setdbprefs` values before each `fetch`.

Examples

Example 1 — Display Current Values

Type `setdbprefs` and MATLAB returns

```
           DataReturnFormat: 'cellarray'
              ErrorHandling: 'store'
             NullNumberRead: 'NaN'
            NullNumberWrite: 'NULL'
             NullStringRead: 'null'
            NullStringWrite: 'null'
         JDBCDataSourceFile: ''
      UseRegistryForSources: 'yes'
   TempDirForRegistryOutput: ''
```

This specifies that

- Data is imported into MATLAB cell arrays.
- Any errors that occur during a connection or an SQL query are stored in the `Message` field of the connection or cursor data object.
- Any `NULL` number in the database is read into MATLAB as `NaN`. Any `NaN` number in MATLAB is exported to the database as a `NULL` number. Any `NULL` string in the database is read into MATLAB as `'null'`. Any `'null'` string in MATLAB is exported to the database as a `NULL` string.
- A MAT-file containing the JDBC source file has not been specified.
- Visual Query Builder will look in the Windows system registry for data sources not found in the `ODBC.INI` file.
- A temporary directory for writing registry settings has not been specified.

Example 2 — Change a Value

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

```
NullNumberRead: 'NaN'
```

This specifies that any NULL number in the database is read into MATLAB as NaN.

To change the value to 0, type

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

This specifies that any NULL number in the database is read into MATLAB as 0.

Example 3 — Change the DataReturnFormat

Cell array: to specify the cellarray format, type

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

This specifies that data is imported into MATLAB cell arrays. The following illustrates a subsequent import.

```
conn = database('SampleDB', '', '');
curs=exec(conn, ...
 'select all ProductName,UnitsInStock fromProducts');
curs=fetch(curs,3);
curs.Data
ans =
    'Chai'             [39]
    'Chang'            [17]
    'Aniseed Syrup'    [13]
```

Numeric: Specify the numeric format by typing

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

Performing the same set of import functions used in the cell array example results in

```
curs.Data
ans =
   NaN    39
```

```
     NaN    17
     NaN    13
```

In the database, the values for ProductName are all character strings, as seen in the previous results when DataReturnFormat is set to cellarray. The ProductName values cannot be read when they are imported using the numeric format. Therefore, MATLAB treats them as NULL numbers and assigns them as NaN, which is the current value for the NullNumberRead property of setdbprefs in this example.

Structure: Specify the structure format by typing

```
setdbprefs ('DataReturnFormat','structure')
```

Performing the same set of import functions used in the cell array example results in

```
curs.Data
ans =
     ProductName: {3x1 cell}
    UnitsInStock: [3x1 double]
```

View the contents of the structure to see the data.

```
curs.Data.ProductName
ans =
    'Chai'
    'Chang'
    'Aniseed Syrup'

curs.Data.UnitsInStock
ans =
    39
    17
    13
```

Example 4 — Change the Write Format for NULL Numbers

To specify the `NullNumberWrite` format as `NaN`, type

```
setdbprefs('NullNumberWrite', 'NaN')
```

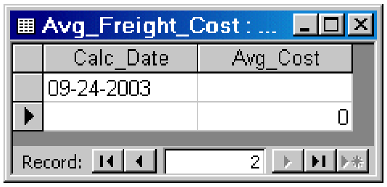
This specifies that any numbers represented as `NaN` in MATLAB are exported to a database as `NULL`.

For example, the variable `ex_data`, contains a `NaN`.

```
ex_data =
      '09-24-2003'         NaN
```

Executing a `fastinsert` for `ex_data` will export the `NaN` as `NULL` as in

```
fastinsert (conn, 'Avg_Freight_Cost', colnames, ex_data)
```



	Calc_Date	Avg_Cost
	09-24-2003	
▶		0

Change the `NullNumberWrite` value to `Inf`.

```
setdbprefs('NullNumberWrite', 'Inf')
```

Attempt to insert `ex_data`, which contains a `NaN`. MATLAB does not recognize the `NaN` in `ex_data` and generates an error.

```
fastinsert(conn, 'Avg_Freight_Cost', colnames, ex_data
??? Error using ==> fastinsert
[Microsoft][ODBC Microsoft Access Driver]
Too few parameters.
Expected 1.
```

Example 5 — Change the ErrorHandling

Store: To specify the `store` format, type

```
setdbprefs ('ErrorHandling','store')
```

This specifies that any errors from running `database` or `exec` are stored in the `Message` field of the returned connection or cursor object.

The following illustrates an example of trying to fetch from a closed cursor with the `store` option for `ErrorHandling`.

```
conn=database('SampleDB', '', '');
curs=exec(conn, 'select all ProductName from Products');
close(curs)
curs=fetch(curs,3);
curs=

        Attributes: []
              Data: 0
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: 'select all ProductName from Products'
           Message: 'Error: Invalid cursor'
              Type: 'Database Cursor Object'
         ResultSet: 0
            Cursor: 0
         Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
             Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

The error indication appears in the `Message` field.

Report: To specify the report format, type

```
setdbprefs ('ErrorHandling','report')
```

This specifies that any errors from running database or exec display immediately in the Command Window.

The following illustrates the same example as above when trying to use fetch from a closed cursor with the report option for ErrorHandling.

```
conn = database('SampleDB', '', '');
curs=exec(conn, 'select all ProductName from Products');
close(curs)
curs=fetch(curs,3);
??? Error using ==> cursor/fetch (errorhandling)
Invalid Cursor
Error in ==>
 D:\matlab\toolbox\database\database\@cursor\fetch.m
On line 36  ==>    errorhandling(initialCursor.Message);
```

The error indication appears immediately in the Command Window.

Empty: To specify the `empty` format, type

```
setdbprefs ('ErrorHandling','empty')
```

This specifies that any errors from running `database` or `exec` are stored in the `Message` field of the returned connection or cursor object. In addition, objects that cannot be created are returned as empty handles, [ ].

The following illustrates the same example as above when trying to use `fetch` from a closed cursor with the `empty` option for `ErrorHandling`.

```
conn = database('SampleDB', '', '');
curs=exec(conn, 'select all ProductName from Products');
close(curs)
curs=fetch(curs,3);
curs =

        Attributes: []
              Data: []
    DatabaseObject: [1x1 database]
          RowLimit: 0
          SQLQuery: 'select all ProductName from Products'
           Message: 'Invalid Cursor'
              Type: 'Database Cursor Object'
         ResultSet: 0
            Cursor: 0
         Statement: [1x1 sun.jdbc.odbc.JdbcOdbcStatement]
     Fetch: [1x1 com.mathworks.toolbox.database.fetchTheData]
```

The error indication appears in the cursor object `Message` field. In addition, the `Attributes` field returned empty handles because no attributes could be created.

Example 6 — Change Multiple Settings

Type

```
setdbprefs({'NullStringRead';'DataReturnFormat'},...
{'NaN';'numeric'})
```

This specifies that any `NULL` string in the database is read into MATLAB as `'NaN'`, and data is retrieved into a matrix of doubles. See also “Example 8 — Assign Values Using Structure” on page 5-124 for another way to change multiple settings.

Example 7 — Specify JDBC Data Sources

Type

```
setdbprefs('JDBCDataSourceFile',...
 'D:/Work/myjdbcdatasources.mat')
```

to instruct VQB to use the data sources specified in the file `myjdbcdatsources.mat`, where `myjdbcdatasources.mat` was defined in VQB using **Query > Define JDBC Data Source**.

Example 8 — Assign Values Using Structure

Assign values for some preferences to fields in the structure `s`

```
s.DataReturnFormat = 'numeric';
s.NullNumberRead = '0';
s.TempDirForRegistryOutput = 'C:\Work'
```

and MATLAB displays

```
s =
            DataReturnFormat: 'numeric'
              NullNumberRead: '0'
    TempDirForRegistryOutput: 'C:\Work'
```

Set the preferences using the values in `s` by running

```
setdbprefs(s)
```

Values for other preferences remain unchanged, and if not specifically assigned, maintain the defaults. For example, run `setdbprefs` and MATLAB displays the values you just assigned as well as the existing values for other preferences:

```
setdbprefs

                DataReturnFormat: 'numeric'
                   ErrorHandling: 'store'
                  NullNumberRead: '0'
                 NullNumberWrite: 'NaN'
                  NullStringRead: 'null'
                 NullStringWrite: 'null'
              JDBCDataSourceFile: ''
           UseRegistryForSources: 'yes'
        TempDirForRegistryOutput: 'C:\Work'
```

Example 9 — Return Values to Structure

You can assign the values for all preferences to `s` by running

```
s = setdbprefs
```

and MATLAB displays, for example

```
s =

                DataReturnFormat: 'cellarray'
                   ErrorHandling: 'store'
                  NullNumberRead: 'NaN'
                 NullNumberWrite: 'NaN'
                  NullStringRead: 'null'
                 NullStringWrite: 'null'
              JDBCDataSourceFile: ''
           UseRegistryForSources: 'yes'
        TempDirForRegistryOutput: ''
```

This allows you to use the MATLAB tab completion feature when obtaining the value for a preference. For example, type

```
s.U
```

and press the **Tab** key, and then **Enter**. MATLAB completes the field and displays the value:

```
s.UseRegistryForSources

ans =

yes
```

Example 10 — Save Preferences for Later Reuse

If you regularly use sets of preferences, consider saving the values to a MAT-file so you can easily assign them at a later time. For example, assume you regularly use a set of preferences for the Seasonal Smoothing project. Assign the variable `SeasonalSmoothing` to the preferences, and save it to the MAT-file `SeasonalSmoothingPrefs` (in the current directory):

```
SeasonalSmoothing = setdbprefs;
save SeasonalSmoothingPrefs.mat SeasonalSmoothing
```

At a later time, you can load the data and restore the preferences:

```
load SeasonalSmoothingPrefs.mat
setdbprefs(SeasonalSmoothing);
```

See Also

`clear`, `cursor.fetch`, `getdatasources`

Purpose Convert JDBC SQL grammar to system's native SQL grammar

Syntax

```
n = sql2native(conn, 'sqlquery')
```

Description `n = sql2native(conn, 'sqlquery')` for the connection `conn`, which was created using `database`, converts the SQL statement string `sqlquery`. The string is converted from JDBC SQL grammar into the database system's native SQL grammar, returning the native SQL statement to `n`.

Purpose

Detect whether property is supported by database metadata object

Syntax

```
a = supports(dbmeta)
a = supports(dbmeta, 'property')
a.property
```

Description

`a = supports(dbmeta)` returns a structure 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 create `a` using `supports`.

There are dozens of properties for `dbmeta`. Examples include `'GroupBy'` and `'StoredProcedures'`.

Examples

Type

```
a = supports(dbmeta, 'GroupBy')
```

and MATLAB returns

```
a =
     1
```

indicating that the database supports the use of SQL group-by clauses.

To find the `GroupBy` value as well as values for all other properties, type

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

MATLAB returns a list of properties and their values. The `GroupBy` property is included in the list. You can also see its value by typing

```
a.GroupBy
```

to which MATLAB returns

```
a =
     1
```

See Also

`database`, `dmd`, `get`, `ping`

tableprivileges

Purpose

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

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`

Purpose

Replace data in database table with data from MATLAB

Syntax

```
update(conn, 'tab', colnames, exdata, 'whereclause')
update(conn, 'tab', colnames, ...
{datA,datAA,...; datB,datBB,...; datn,datnn}, ...
{'where col1 = val1'; where col2 = val2'; ... 'where  coln = valn'}
```

Description

`update(conn, 'tab', colnames, exdata, 'whereclause')` exports data from the MATLAB variable `exdata`, into the database table `tab`, via the database connection `conn`. The variable `exdata` can be a cell array, numeric matrix, or structure. You do not define the type of data you are exporting; the data is exported in its current MATLAB format. Existing records in the table are replaced as specified by the SQL command `whereclause`. Specify the column names for `tab` as strings in the MATLAB cell array, `colnames`. If `exdata` is a structure, field names in the structure must exactly match `colnames`.

The status of the `AutoCommit` flag determines if `update` automatically commits the data or if a `commit` is needed. View the `AutoCommit` flag status for the connection using `get` and change it using `set`. Commit the data using `commit` or issue an SQL commit statement via the `exec` function. Roll back the data using `rollback` or issue an SQL rollback statement via the `exec` function.

To add new rows instead of replacing existing data, use `fastinsert`.

`update(conn, 'tab', colnames, {datA,datAA,...; datB,datBB,...; datn,datnn}, {'where col1 = val1'; where col2 = val2'; ...  'where coln = valn'})` exports multiple records based on `n` different `where` clauses. The number of `where` clauses must equal `n`, the number of records in `exdata, n`.

Remarks

Do not count on the order of records in your database as being constant, but rather always use the values in column names to identify records.

If you get an error, it might be because the table is open in design mode in Access (edit mode for other databases). Close the table in the database and repeat the `fastinsert` function. For example, the error might be

```
[Vendor][ODBC Product Driver] The database engine could
not lock table 'TableName' because it is already in use
by another person or process.
```

If you get this error

```
??? Error using ==> database.update
Error:Commit/Rollback Problems
```

it could be because you are trying to perform an update identical to one you just performed.

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.

Example 3 — Update Multiple Records Using Different Constraints

Given the following data in the table `TeamLeagues`, where the column names are `'Team'`, `'Zip_Code'`, and `'New_League'`,

```
'Team1'    02116
'Team2'    02138
'Team3'    02116
```

assign teams with a zip code of `02116` to the `A` league and teams with a zip code of `02138` to the `B` league:

```
update(conn, 'TeamLeagues', {'League'}, {'A';'B'}, ...
{'where Zip_Code =''02116''';'where Zip_Code =''02138'''})
```

See Also

`commit`, `database`, `fastinsert`, `rollback`, `set`

versioncolumns

Purpose

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 updated.

See Also

`columns`, `dmd`, `get`

Purpose

Field size of column in fetched data set

Syntax

`colsize = width(cursor, colnum)`

Description

`colsize = width(cursor, colnum)` returns the field size of the specified column number `colnum`, in the fetched data set `curs`.

Examples

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`, `cursor.fetch`, `get`

Examples

Use this list to find examples in the documentation.

Setting Up a Data Source

Visual Query Builder GUI: Importing Data

Visual Query Builder GUI: Displaying Results

Visual Query Builder GUI: Exporting Data

Using Database Toolbox Functions

Index