Database Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

User's Guide

Version 1

How to Contact The MathWorks:

508-647-7000	Phone

508-647-7001	Fax

The MathWorks, Inc. 24 Prime Park Way Natick, MA 01760-1500	Mail

`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

Database Toolbox User's Guide

Printing History: May 1998 Version 1 release for MATLAB 5.2 (online only)
July 1998 First printing for Version 1 (Releases 10 and 11)

Contents

1 Introduction

2 Installation and Setup

3 Tutorial

4 Reference

1

Introduction

What Is the Database Toolbox?

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

With the Database Toolbox, you can bring data from an existing database into MATLAB, use any of MATLAB's computational and analytic tools, and store the results back in the database or in another database. You import the data into MATLAB by reading data files and then save the MATLAB data in files.

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.

The Database Toolbox connects MATLAB to a database using MATLAB commands. Data is retrieved from the database as a string, parsed into the correct data types, and stored in a MATLAB cell array. At that point, you use MATLAB's extensive set of tools to work with the data. You can include Database Toolbox commands in MATLAB M-files. To export the data from MATLAB to a database you use MATLAB commands.

Features of the Database Toolbox

The Database Toolbox has the following features:

- Data types are automatically preserved in MATLAB – No data massaging or manipulation is required. The data is stored in MATLAB cell arrays, which support mixed data types.
- Different databases can be used in a single session – Import data from one database, perform calculations, and export the modified or unmodified data to another database. Multiple databases can be open during a session.
- Dynamic importing of data from within MATLAB – Modify your SQL queries in MATLAB statements to retrieve the data you need.
- Single environment for faster data analysis – Access both database data and MATLAB functions at the MATLAB command prompt.
- Database connections remain open until explicitly closed – Once the connection to a database has been established, it remains open during the entire MATLAB session until you explicitly close it. This improves database

access and reduces the number of commands necessary to import/export data.

- Multiple cursors supported for a single database connection – Once a connection has been established with a database, the connection can support the use of multiple cursors. You can execute several queries on the same connection.
- Retrieval of large data sets or partial data sets – You can retrieve large data sets from a database in a single fetch or in discrete amounts using multiple fetches.

How To Use This Book

This book describes how to install and use the Database Toolbox.

Who Should Read This Book

This book assumes that you have a working understanding of MATLAB as well as the Structured Query Language (SQL) and the database applications you use.

The Database Toolbox uses MATLAB cell arrays; if you need to know more about MATLAB cell arrays, see Chapter 13 in *Using MATLAB*.

How This Book Is Organized

This book contains four chapters:

Chapter 1, Introduction, provides an overview of the Database Toolbox and of this book.

Chapter 2, Installation and Setup, provides system requirements and describes how to install the Database Toolbox and set up an ODBC data source or a JDBC driver.

Chapter 3, Tutorial, presents examples with instructions for using most toolbox commands. The tutorial uses a sample database, `Northwind`, that is distributed with Microsoft Access. If you have Microsoft Access installed on your system, you can perform the steps exactly as shown. One example uses a different database, `tutorial`, an Access database that is distributed with the Database Toolbox.

Chapter 4, Command Reference, is an alphabetical reference of all commands in the toolbox.

Documentation Conventions

This book uses the following typographical conventions.

To Indicate	This Guide Uses	Example
Example code	`Monospace type`	To assign the value 5 to A, enter `A = 5`
Command names and syntax	`Monospace type`	The `close` command uses the syntax: `close(cursor)`
Keys	**Boldface** with an initial capital letter	Press the **Enter** key.
Mathematical expressions	Variables in *italics*. Functions, operators, and constants in standard type.	This vector represents the polynomial $p = x^2 + 2x + 3$

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

Menu names, menu items, and controls	**Boldface** with an initial capital letter	Choose the **File** menu.
New terms	*Italics*	An *array* is an ordered collection of information.

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

```
get(connection, 'autocommit')
```

Other Relevant Books

MATLAB comes with an extensive set of documentation consisting of an online Help facility, an online *Function Reference,* and printed manuals. The full set of printed documentation includes the following titles:

- The *MATLAB Installation Guide* describes how to install MATLAB on your platform.
- *Getting Started with MATLAB* describes MATLAB fundamentals to beginning MATLAB users.
- *Using MATLAB* describes in-depth material on the MATLAB language, working environment, and mathematical topics.
- *Using MATLAB Graphics* describes how to use MATLAB's graphics and visualization tools.
- The *MATLAB Application Program Interface Guide* describes how to write C or Fortran programs that interact with MATLAB.
- User's guides for toolboxes.

2

Installation and Setup

System Requirements

The Database Toolbox works with the following systems and applications.

Platforms

The Database Toolbox runs on the following platforms:

- Microsoft Windows NT 4.0
- Microsoft Windows 95

MATLAB Version

The Database Toolbox requires MATLAB Version 5.2 (R10) or later. You can see the system requirements for MATLAB online at `http://www.mathworks.com/products/ml5_sysreq.shtml`.

Databases

Your system must have access to an installed database. The Database Toolbox supports import/export of data for the following database management systems:

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

Drivers

The Database Toolbox supports all Open Database Connectivity (ODBC) drivers used with the supported databases. Java Database Connectivity (JDBC) drivers are *not* officially supported, but some have been tested and found workable.

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

About Drivers for the Database Toolbox

An ODBC driver is a standard interface that enables communication between database management systems and SQL-based applications. Every database supported by the Database Toolbox provides an ODBC driver.

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

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

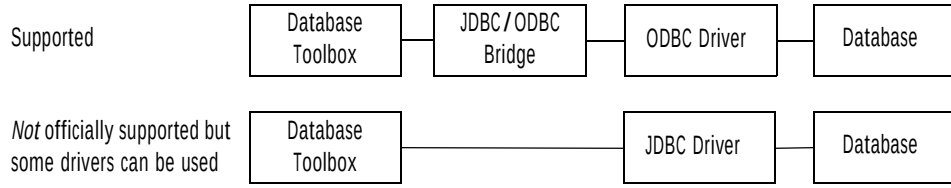
Third-party companies supply JDBC drivers. It is technically possible to use a JDBC driver with the Database Toolbox instead of using an ODBC driver, but JDBC drivers are not officially supported by The MathWorks.

The MathWorks has tested the JDBC drivers listed below and found them to be usable with the Database Toolbox. You may find that you can use them without any problems.

Table 2-1: Tested JDBC Drivers

Driver	Database	Company
ORACLE	ORACLE	ORACLE
FastForward	SYBASE MS SQL	Connect Software
Imaginary	MS SQL	Imaginary
Kona	SYBASE	KonaSoft, Inc.

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



Structured Query Language (SQL)

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

Data Types

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

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

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

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

Installing the Database Toolbox

To install the Database Toolbox, select it with any other MATLAB toolboxes you want to install when you install MATLAB. See the *MATLAB Installation Guide* for your platform for more information.

Setting Up a Data Source

Note: If you are using a JDBC driver, you do not need to set up a data source. Instead you provide the JDBC driver with the necessary database location information when you connect to the database using the `database` command.

To use the Database Toolbox with your database and an ODBC driver, you need to set up a *data source*. A data source consists of data that you want the toolbox to access and information on how to find the data, such as driver, directory, server, or network names. You assign a name to each data source.

You can set up local data sources for databases that reside on your PC, or remote data sources for databases that reside on systems that are networked to your PC. The data source setup procedures for local and remote data sources differ slightly – use the appropriate instructions for your database.

Setting Up a Local Data Source

Follow this procedure to set up a local data source. This procedure uses as an example, the Microsoft ODBC driver Version 3.40 and Microsoft Access Version 7.00 for Windows NT. If you are using a different configuration, you will have to modify the instructions slightly.

If you have Microsoft Access installed and want to use the tutorial in Chapter 3 as written, follow information labeled “For the tutorial” when you set up the data source. The tutorial uses the data source, `SampleDB`, which you set up here. `SampleDB` uses the Microsoft Access sample database called `Northwind`.

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

   The **ODBC Data Source Administrator** dialog box appears, listing any existing data sources.
3. Click **Add**. A list of installed ODBC drivers appears in the **Create New Data Source** dialog box.

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

- For the tutorial, select **Microsoft Access Driver**.
- Otherwise, select the driver for your database.

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

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

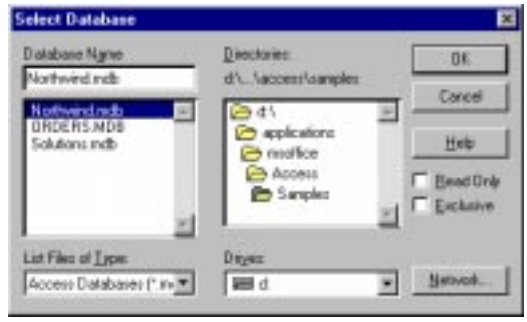
For the tutorial, type `SampleDB` as the data source name, and for the description, type `For the Database Toolbox tutorial`.

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

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

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

      The **Select Database** dialog box appears.



   b Find and select the database you want to use.

      For the tutorial, select `Northwind.mdb` in the `msoffice\Access\Samples` directory.

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

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

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

Setting Up a Remote Data Source

Follow this procedure to set up a data source that resides on a remote system to which your PC has network access. This procedure uses the Microsoft ODBC driver Version 3.40 and Microsoft Access Version 7.00 for Windows NT that is installed on a network server. If you are using a different configuration, you will have to modify the instructions slightly.

If you have Microsoft Access installed on a networked server and want to use the tutorial in Chapter 3 as written, follow information labeled “For the tutorial” when you set up the data source. The tutorial uses the data source,

`SampleDB`, which you set up here. `SampleDB` uses the Microsoft Access sample database called `Northwind`.

1 From the Windows **Start** menu, select **Control Panel** from the **Settings** menu.

2 Double-click **ODBC**.

The **ODBC Data Source Administrator** dialog box appears.

3 Click the **System DSN** tab.

A list of existing system data sources appears.

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

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

- For the tutorial, select **Microsoft Access Driver**.
- Otherwise, select the driver for your database.

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

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

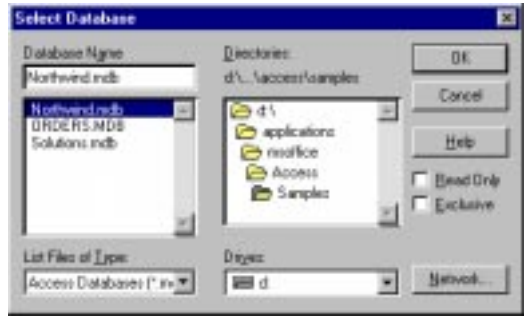
For the tutorial, type `SampleDB - Remote` as the data source name, and for the description, type `For the Database Toolbox tutorial`.

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

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

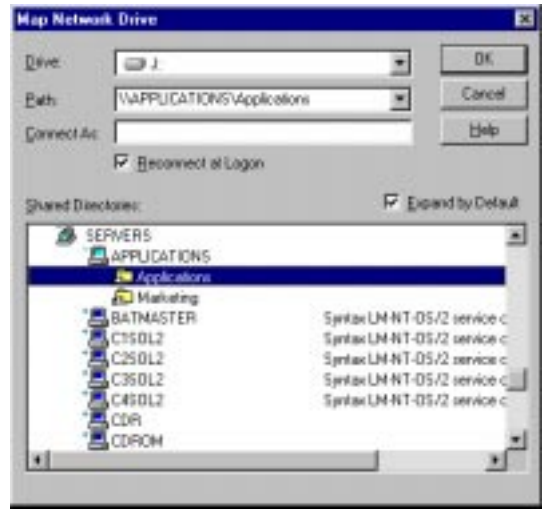
a In the **ODBC Setup** dialog box, click **Select**.

The **Select Database** dialog box appears.



b Click **Network**.

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



c Find and select the database you want to use. For the tutorial, select `Northwind.mdb` in the `msoffice\Access\Samples` directory.

In this example, the database is in `SERVERS\APPLICATIONS\Applications`. After selecting this directory, the **Map Network Drive** dialog box closes automatically and you continue locating the file in the **Select Database** dialog box.

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

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

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

3

Tutorial

Introduction

This tutorial demonstrates many of the Database Toolbox commands using simple examples. It uses Microsoft Access as the sample database application. The first four examples use the `Northwind` database that is included in the Access samples. The last example uses the Access `tutorial` database that ships with the Database Toolbox.

The tutorial consists of five parts:

- Importing data into MATLAB from a database
- Viewing information about the imported data
- Exporting data from MATLAB to a new record in a database
- Exporting data from MATLAB, replacing existing data in a database
- Exporting multiple records from MATLAB

M-files containing commands used in these tutorials are in the `matlab\toolbox\database\dbdemos` directory. As you work with the tutorials in this chapter, you can open the M-files to see the commands and copy them, or you can run the M-files to see the results.

For more information on the commands used in this tutorial, see Chapter 4.

Importing Data into MATLAB from a Database

In this part of the tutorial, you connect to and import data from a database. Specifically, you connect to the `SampleDB` data source and then import `country` data from the `customers` table in the `Northwind` sample database. You use these Database Toolbox commands:

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

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

1 If you did not already do so, set up the data source according to the directions in "Setting Up a Data Source" on page 2-6.

Follow the instructions marked "For the tutorial." The data source you create, `SampleDB`, is used for the tutorial. `SampleDB` uses `Northwind`, a sample database distributed with Microsoft Access.

2 In MATLAB, set the maximum time, in seconds, that will be allowed for the MATLAB session to connect to a database successfully. This prevents the MATLAB session from hanging up if a database connection fails.

You must enter the command before you connect to a database.

Type

```
logintimeout(5)

ans=
     5
```

to specify the maximum allowable connection time as five seconds. If you are using a JDBC connection, the command syntax is different – for more information, see `logintimeout` in Chapter 4.

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

3 Connect to a database – type:

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

In this example, you define a MATLAB variable, `connA`, to be the returned connection structure. This connection stays open until you close it with the `close` command.

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

If you are using a JDBC connection, the command syntax is different – for more information, see the `database` reference pages.

4 Check the connection status – type:

```
ping(connA)
```

MATLAB returns information about the connection, indicating that the connection was successful:

```
Database Product Name     : ACCESS
Database Product Version  : 3.0
JDBC Driver Name          : JDBC-ODBC Bridge (odbcjt32.dll)
JDBC Driver Version       : 1.1001 (3.40.2829)
Max. Database Connections : 64
Current User Name         : admin
Database URL              : jdbc:odbc:SampleDB
Auto Commit Transactions  : True
```

5 Open a cursor and execute an SQL statement – type:

```
cursorA = exec(connA, 'select country from customers')
```

A cursor structure is returned and stored in a MATLAB cell array (cell arrays support mixed data types). In this example, you assign the MATLAB variable `cursorA` to the returned cursor structure.

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

6 Import data into MATLAB – type:

```
cursorA = fetch(cursorA, 10)
```

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

- `cursorA`, the cursor structure returned by `exec`.
- `10`, the maximum number of rows you want to be returned by `fetch`. The `rowlimit` argument is optional and if not included, MATLAB imports all remaining rows.

In this example, `fetch` reassigns the variable `cursorA` to the cursor structure containing the rows of data returned by `fetch`.

7 Display the data element in the cell array for cursorA – type:

```
AA = cursorA.data
```

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

The cursorA cell array contains an element, data, that points to the rows of data in the array. Here you assign the variable AA to the data element, cursorA.data.

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

```
close(cursorA)
close(connA)
```

Viewing Information About the Imported Data

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

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

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

1 If you are continuing directly from the previous part of the tutorial, skip this step. Otherwise, if the cursor and connection are not open, type the following to continue with this tutorial.

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

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

```
rowsA = rows(cursorA)

rowsA =
    10
```

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

3 View the number of columns in the data set – type:

```
columnsA = cols(cursorA)

columnsA =
     1
```

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

4 View the column names for the columns in the data set – type:

```
colnamesA = columnnames(cursorA)

colnamesA =
'country'
```

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

5 View the width of the column (size of field) in the data set – type:

```
widthA = width(cursorA, 1)

widthA =
    15
```

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

6 You can use a single command to view multiple attributes for a column – type:

```
attrA = attr(cursorA)

attrA =
      columnName: 'country'
       typeName: 'TEXT'
      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 `column` argument to specify the column for which you want the information.

7 Close the cursor – type:

```
close(cursorA)
```

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

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

```
close(connA)
```

Exporting Data from MATLAB to a New Record in a Database

In this part of the tutorial, you retrieve a set of data, perform a simple calculation on the data using MATLAB, and export the results as a new record to another table in the database. Specifically, you retrieve freight costs from an orders table, calculate the average freight cost, put the data into a cell array to export it, and then export the data (the average freight value and the number of shipments on which the average was based) to an empty table.

You use these Database Toolbox commands:

- `get`
- `insert`

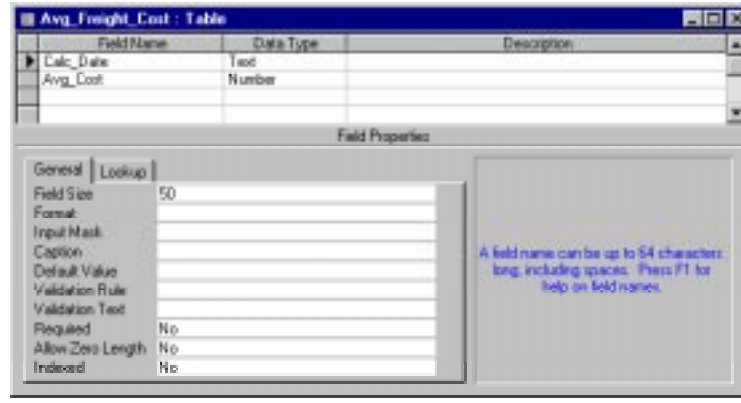
If you want to see or copy the commands for this part of the tutorial, or if you want to run the set of commands, use the M-file `matlab\toolbox\database\dbdemos\dbinsertdemo.m`.

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

Open the `Northwind` database in Access. Create a new table called `Avg_Freight_Cost` that has two columns, `Calc_Date` and `Avg_Cost`. 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**. Be sure to close the table; Access then warns you that there is no primary key, but you do not need one.

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

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



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

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

3 In MATLAB, import the data on which you will perform calculations. Specifically, import the `freight` column of data from the `orders` table. To keep the example simple, import only three rows of data. Type:

```
cursorA = exec(connA, 'select freight from orders');
cursorA = fetch(cursorA, 3)
```

4 View the data you imported – type:

```
AA = cursorA.data
```

MATLAB returns:

```
AA =
    [32.3800]
    [11.6100]
    [65.8300]
```

5 Calculate the average freight cost. First, assign the variable name `rowsA` to the number of rows in the array. Then convert the cell array to a vector and calculate the average, assigning the result to the variable `meanA`.

```
rowsA = rows(cursorA);
meanA = sum([AA{:}])/rowsA

meanA =
   36.6067
```

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

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

D =
1/20/98
```

7 Create a cell array that will contain the data you are exporting. This example assigns the name `C` to the cell array and defines `C` as being 1 row by 2 columns.

```
C = cell(1,2)

C =
     []     []
```

8 Assign the date to the first cell and the mean to the second cell.

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

C =
    '1/20/98'    []

C(1,2) = {meanA}

C =
    '1/20/98'    [36.6067]
```

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

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

colnames =
    'Calc_Date'    'Avg_Cost'
```

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

Verify the status of the `autocommit` flag using the `get` command - type:

```
get(connA, 'autocommit')

ans =
on
```

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

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

```
insert(connA, 'Avg_Freight_Cost', colnames, C)
```

`connA` is the connection structure for the database to which you are exporting data. In this example, the database to which you are exporting data is `SampleDB`, for which there is already an open connection, `connA`. However, if you are exporting to a different database, use the `database` command to connect to it before exporting the data.

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

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

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

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

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

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

13 Close the cursor – type:

```
close(cursorA)
```

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

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

```
close(connA)
```

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

Exporting Data from MATLAB, Replacing Existing Data in a Database

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

You use these Database Toolbox commands:

- `close`
- `update`

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

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

```
connA = database('SampleDB', '', '');
colnames = {'Calc_Date', 'Avg_Cost'}
C = cell(1,2);
D = '1/20/98';
meanA = 36.6067;
C = {D, meanA}
```

MATLAB returns:

```
C =
    '1/20/98'    [36.6067]
```

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

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

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

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

C =
    '1/19/98'    [36.6067]
```

4 Identify the record to be updated in the database. To do so, define an SQL `where` statement and assign it to the variable `whereclause`. The record to be updated is the record that has `1/20/98` for the `Calc_Date`.

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

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

Because the date string is within a string, two single quotes surround the date instead of the usual single quote.

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

```
update(connA, 'Avg_Freight_Cost', colnames, C, whereclause)
```

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

7 Close the cursor and disconnect from the database.

```
close(cursorA)
close(conn)
```

Always close a cursor and a connection when you are finished with them to avoid using memory unnecessarily and to ensure there are enough available connections and cursors for other users. Close the cursor before closing the connection.

Exporting Multiple Records from MATLAB

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

You use this Database Toolbox command:

- `insert`

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

1. Set up a data source for the Microsoft Access `tutorial` database that shipped with the Database Toolbox. It is in the `matlab\toolbox\database\dbdemos` directory. Refer to the directions in "Setting Up a Data Source" on page 2-6. Name the data source `tutorial` and for the description, type `Tutorial for inserting multiple records`.

2. Connect to the database – type:

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

   You define the returned connection structure as `conn`. You do not need a user name or password to access the `tutorial` database.

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

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

4. To get a sense of the data you imported, view the column names in the fetched data set – type:

   ```
   columnnames(curs)

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

5 To get a sense of what the data is, view the data for January, which is in column two – type:

```
curs.data(:,2)

ans =
        5000
        3000
        2500
        4300
        1200
        3500
        1800
        2400
        1400
        1375
```

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

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

m =
    10
n =
    13
```

7 Compute the monthly totals – type:

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

This creates a column of data, `monthly`, in which each row is the total sales volume of all products for that month. For example, when `i` is `2`, row `1` of

`monthly` is the total of all rows in column 2 of `curs.data`, column 2 being the sales volume for January.

The result is:

```
monthly =
       26475
       18621
       14606
       11944
        9965
        8643
        6525
        5899
        8632
       13170
       48345
      172000
```

8 In order to export the column of data, you must first convert it to a cell array – type:

```
monthlyTotals = num2cell(monthly)
```

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

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

```
colNames{1,1} = 'salesTotal'
```

10 Insert the data into the `yearlySales` table – type:

```
insert(conn, 'yearlySales', colNames, monthlyTotals)
```

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

12 Close the cursor and disconnect from the database.

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

4

Reference

Commands Grouped by Purpose

The tables below group Database Toolbox commands by purpose.

Database Connection

Command	Purpose
`close`	Close database connection.
`database`	Connect to database.
`get`	Get property of database connection.
`logintimeout`	Set or get time allowed to establish database connection.
`ping`	Get information about database connection.
`set`	Set `autocommit` flag for database.

SQL Cursor

Command	Purpose
`close`	Close database cursor.
`exec`	Execute SQL statement and open cursor.
`get`	Get property of database connection.
`querytimeout`	Get time allowed for database SQL query to succeed.
`set`	Set `rowlimit` for `fetch`.

Importing Data into MATLAB from a Database

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

Exporting Data from MATLAB to a Database

Command	Purpose
insert	Export MATLAB cell array data into database table.
update	Replace data in database table with data from MATLAB cell array.

Commands in Alphabetical Order

This section contains detailed descriptions of all Database Toolbox commands. You can also access this information through the online Help Desk.

Purpose

Get attributes of columns in fetched data set

Syntax

```
attributes = attr(cursor, column)
attributes = attr(cursor)
```

Description

`attributes = attr(cursor, column)` gets attribute information for the specified `column` in the fetched data set, `cursor`.

`attributes = attr(cursor)` gets attribute information for all columns in the fetched data set, `cursor`, and stores it in a cell array. Use `attributes(n)` to display the attributes for column `n`.

The returned attributes are:

`columnName`	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(cursorA, 4)

ans =
      columnName: '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 `cursorA`, and assign them to `attrA`:

```
attrA = attr(cursorA)
```

View the attributes of column 4:

```
attrA(4)
```

MATLAB returns the attributes of column 4:

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

See Also

`cols`, `columnnames`, `fetch`, `width`

Purpose

Close database connection or cursor

Syntax

```
close(conn)
close(cursor)
```

Description

`close(conn)` closes the database connection, `conn`.

`close(cursor)` closes the cursor structure, `cursor`, that was returned by `fetch` or `exec`.

Database connections and cursors remain open until you close them using the `close` command. Always close a cursor or connection 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 a cursor and connection 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 either of the overloaded methods:

```
help database/close.m
help cursor/close.m
```

Examples

To close the cursor `curs1` and the connection `connA`, type:

```
close(curs1);
close(connA)
```

See Also

`database`, `exec`, `fetch`

cols

Purpose

Get number of columns in fetched data set

Syntax

```
numcols = cols(cursor)
```

Description

`numcols = cols(cursor)` gets the number of columns in the fetched data set, `cursor`.

Examples

This example shows that there are three columns in the fetched data set, `cursorA`:

```
colsA = cols(cursorA)

colsA =
     3
```

See Also

`columnnames`, `fetch`, `rows`, `width`

Purpose

Get names of columns in fetched data set

Syntax

```
colnames = columnnames(cursor)
```

Description

`colnames = columnnames(cursor)` gets the column names in the fetched data set, `cursor`. The column names are returned as a single string vector.

Examples

The fetched data set, `cursorA`, contains three columns having the names shown:

```
colsA = columnnames(cursorA)

colsA =

    'Address', 'City', 'Country'
```

See Also

`attr`, `cols`, `fetch`, `width`

database

Purpose

Connect to database

Syntax

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

Description

`conn = database('datasourcename', 'username', 'password')` connects a MATLAB session to a database via an ODBC driver, returning the connection structure to `conn`. `datasourcename` is the data source to which you are connecting. You must have previously set up the data source – for instructions, see "Setting Up a Data Source" on page 2-6. `username` and `password` are the user name and/or password required to connect to the database. If you do not need a user name or a password to connect to the database, use empty strings as the arguments.

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

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` command to view information about the connection, and use `get` to view properties of `conn`.

The database connection stays open until you close it using the `close` command. 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:

```
connA = database('Pricing', 'mike', 'bravo')
```

Example 2 – Establish ODBC connection without username and password

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

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

Example 3 – Establish JDBC connection

In the JDBC connection example below, the database is `oracle`, the user name is `scott`, and the password is `tiger`. The JDBC driver name is `oracle.jdbc.driver.OracleDriver` and the URL to the database is `jdbc:oracle:oci7:`.

```
conn1 = database('oracle','scott','tiger',...
   'oracle.jdbc.driver.OracleDriver','jdbc:oracle:oci7:')
```

Use the `ping` command to see information about the connection – type `ping(conn1)` and MATLAB returns:

```
Database Product Name     : Oracle
Database Product Version  : Oracle7 Server Release 7.3.3.0.0 -
                            Production Release With the
                            distributed, replication and parallel
                            query options PL/SQL Release
                            2.3.3.0.0 - Production
JDBC Driver Name          : Oracle JDBC driver
JDBC Driver Version       : 7.3.3.1.3
Max. Database Connections : 0
Current User Name         : scott
Database URL              : jdbc:oracle:oci7:oracle
Auto Commit Transactions  : True
```

See Also

`close`, `get`, `logintimeout`, `ping`

exec

Purpose

Execute SQL statement and open cursor

Syntax

```
cursor = exec(conn, 'sqlquery')
```

Description

`cursor = exec(conn, 'sqlquery')` executes a valid SQL statement, `sqlquery`, against the database connection, `conn` and opens a cursor. `exec` returns the cursor structure to the variable, `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.

After opening a cursor, use `fetch` to import data from the cursor.

A cursor stays open until you close it using the `close` command. Always close a cursor after you finish using it.

Examples

Example 1 – Select all data from database table

Select all data from the `customers` table accessed via `connA`. Assign the variable `cursA` to the returned cursor structure:

```
cursA = exec(connA, 'select * from customers')
```

Example 2 – Select one column of data from database table

Select `country` data from the `customers` table accessed via `conn`. Assign the variable `str` to the SQL statement and assign `curs` to the returned cursor:

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

Example 3 – Roll back or commit data exported to database table

Use `exec` to roll back or commit data after running an `insert` or an `update` for which the `autocommit` flag was `off`. To roll back data for `connect`, type:

```
exec(connect, 'rollback')
```

To commit the data, type:

```
exec(connect, 'commit')
```

See Also

`close`, `database`, `fetch`, `querytimeout`, `set`

Purpose

Import data into MATLAB cell array

Syntax

```
cursor = fetch(cursor, rowlimit)
cursor = fetch(cursor)
cursor.data
```

Description

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

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

`cursor.data` points to the element in the structure (cell array) `cursor` that contains the data returned by `fetch`. The data types are preserved (cell arrays support mixed data types). After running `fetch`, display the returned data by typing `cursor.data`.

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

Examples

Example 1 – Import all rows of data

Import all of the data into the cursor structure, `cursorA`:

```
cursorA = fetch(cursorA)
```

The data is put in a cell array pointed to by the element `cursorA.data` of the cursor structure. To display data in the cell array `cursorA.data`, type:

```
cursorA.data
```

MATLAB returns all of the data, which in this example consists of three columns and two rows:

```
ans =

'Sales'       [380344] [100765]
'Operations'  [289034] [100896]
```

Example 2 – Import specified number of rows of data

Specify the `rowlimit` argument to retrieve the first 100 rows of data:

```
curs1 = fetch(curs1, 100)
```

Entering the command again returns the second 100 rows of data:

```
curs1 = fetch(curs1, 100)
```

See Also

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

Purpose

Get property of database connection or cursor structure

Syntax

```
value = get(conn, 'propertyname')
value = get(cursor, 'propertyname')
value = conn.propertyname
value = cursor.propertyname
```

Description

`value = get(conn, 'propertyname')` gets the value of `propertyname` for the database connection, `conn`.

Allowable property names and returned values for `conn` are:

`autocommit`	status of the `autocommit` flag, either `on` or `off`, as specified by `set`
`driver`	driver used for the JDBC connection, as specified by `database`
`handle`	identifying number for the connection
`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`
`URL`	for a JDBC connection only, the JDBC URL structure, `jdbc:subprotocol:subname`, as specified by `database`
`username`	user name required to connect to the database, as specified by `database`; note that you cannot use `get` to retrieve `password`

`value = get(cursor, 'propertyname')` gets the value of `propertyname` for the cursor structure, `cursor`.

Allowable property names and values for `cursor` are:

`attributes`	cursor attributes
`data`	data in the cursor structure data element
`message`	error message returned from `exec` or `fetch`
`rowlimit`	maximum number of rows to be returned by `fetch`, as specified by `set`
`sqlquery`	SQL statement for the cursor, as specified by `exec`

`value = conn.propertyname` gets the value for the specified connection `propertyname`.

`value = cursor.propertyname` gets the value for the specified cursor `propertyname`.

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

```
help cursor/get.m
help database/get.m
```

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 `AA`:

```
connA = database('SampleDB', '', '');
AA = get(connA, 'instance')
```

MATLAB returns:

```
AA =
SampleDB
```

Example 2 – Get a connection property, autocommit flag status

Determine the status of the `autocommit` flag for `connA`:

```
get(connA, 'autocommit')

ans =
on
```

Example 3 – Display data in cursor

Display the data in `cursorA`:

```
get(cursorA, 'data')
```

or type:

```
cursorA.data
```

MATLAB returns:

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

`cursorA` contains one column with four records.

See Also

`attr`, `cols`, `columnnames`, `database`, `fetch`, `rows`, `width`, `set`

insert

Purpose

Export MATLAB cell array data into database table

Syntax

```
insert(conn, 'tablename', colnames, data)
```

Description

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

The status of the `autocommit` flag determines if `insert` automatically commits the data or if an SQL `commit` command is needed following the insert. View the `autocommit` flag status for the connection using `get` and change it using `set`. Perform an SQL `commit` or `rollback` using `exec`.

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

Examples

Example 1 – Insert a record

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

Create a cell array that will contain the record to be inserted. It is one row by two columns:

```
T = cell(1,2)
```

Assign the data to the cell array:

```
T = {'San Diego', 88}
```

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

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

Perform the insert:

```
insert(conn, 'Temperatures', colnamesT, T)
```

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

Example 2 – Insert multiple records

Insert a cell array, `G`, containing 28 rows of data with three columns, into the `Growth` table. The data columns are `Birthdate`, `Avg_Length`, and `Avg_Weight`. The database connection is `conn1`.

Assign the column name strings to a cell array named `colnamesG`:

```
colnamesG = {'Birthdate', 'Avg_Length', 'Avg_Weight'}
```

Insert the data:

```
insert(conn1, 'Growth', colnamesG, G)
```

The records are inserted in the table.

Example 3 – Import records, perform computations, and then export data

Perform calculations on imported data and then export the data. First import all of the data in the `products` table:

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

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

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

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

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

Calculate the discounted price and assign it to the variable `sale_price`:

```
sale_price =.75*price
```

In order to export the data, you must convert it to a cell array. To convert the columns of data into cell arrays, type:

```
id = num2cell(id);
price = num2cell(price);
sale_price = num2cell(sale_price);
```

Create an array, data, that contains the three columns of data to be exported. Put the id data in column one, price in column two, and sale_price in column three:

```
data = id(:,1);
data(:,2) = price;
data(:,3) = sale_price;
```

Assign the column names to a string array, colnames:

```
colnames={'product_id', 'price', 'sale_price'};
```

Export the data to the Sale table:

```
insert(conn, 'Sale', colnames, data)
```

All rows of data are inserted into the Sale table.

Example 4 – Insert followed by SQL commit

This example demonstrates the use of the SQL commit command following an insert. The autocommit flag is off.

Insert the cell array data1 into the column names C of the Error_Rate table:

```
insert(connect, 'Error_Rate', C, data1)
```

Commit the data:

```
cursorset = exec(connect,'commit')
```

See Also database, exec, set, update

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 `database`. 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 `database`. If MATLAB cannot connect within the allowed time, it stops trying.

`timeout = logintimeout('driver')` returns the `time` you set previously using `logintimeout` for the JDBC connection specified by `driver`. A returned value of zero means that the timeout value has not been set previously; MATLAB stops trying to make a connection if it is not successful immediately.

`timeout = logintimeout` returns the `time` you set previously using `logintimeout` for an ODBC connection. A returned value of zero means that the timeout value has not been set previously; MATLAB stops trying to make a connection if it is not successful immediately.

If you do not use `logintimeout` and MATLAB tries to connect without success, your MATLAB session could hang up.

Examples

Example 1 – Get timeout value for ODBC connection

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

```
logintimeout
```

MATLAB returns:

```
ans =

     0
```

The timeout value has not been set.

Example 2 – Set timeout value for ODBC connection

Set the timeout value to five seconds for an ODBC driver:

```
logintimeout(5)

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:

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

ans =
     0
```

The timeout value has not been set. Set the timeout to 10 seconds:

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

timeoutA =
     10
```

Verify the timeout value for the JDBC driver:

```
timeoutA
```

MATAB returns:

```
timeoutA =
     10
```

See Also

`database`

Purpose Get information about database connection

Syntax `ping(conn)`

Description `ping(conn)` returns the status of the database connection, conn. If the connection is open, ping returns information about the connection.

Examples

Example 1 – Get information about ODBC connection

connA is a valid ODBC connection:

```
ping(connA)

Database Product Name     : ACCESS
Database Product Version  : 3.0
JDBC Driver Name          : JDBC-ODBC Bridge (odbcjt32.dll)
JDBC Driver Version       : 1.1001 (3.40.2829)
Max. Database Connections : 64
Current User Name         : admin
Database URL              : jdbc:odbc:SampleDB
Auto Commit Transactions  : True
```

Example 2 – Get information about JDBC connection

connection is a valid JDBC connection:

```
ping(connection)

Database Product Name     : Oracle
Database Product Version  : Oracle7 Server Release 7.3.3.0.0 -
                            Production Release With the
                            distributed, replication, parallel
                            query and Spatial Data options PL/SQL
                            Release 2.3.3.0.0 - Production
JDBC Driver Name          : Oracle JDBC driver
JDBC Driver Version       : 7.3.3.1.3
Max. Database Connections : 0
Current User Name         : scott
Database URL              : jdbc:oracle:oci7:
Auto Commit Transactions  : True
```

Example 3 – Unsuccessful request for information about connection

connB has been terminated or was not successful:

```
ping(connB)

Cannot Ping the Database Connection
```

See Also

database, set

Purpose

Get time allowed for database SQL query to succeed

Syntax

```
timeout = querytimeout(cursor)
```

Description

`timeout = querytimeout(cursor)` gets the amount of time, in seconds, allowed for an SQL query of `cursor`, which is run using `exec`, to succeed. 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 `cursorA`:

```
querytimeout(cursorA)

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`

rows

Purpose

Get number of rows in fetched data set

Syntax

```
numrows = rows(cursor)
```

Description

`numrows = rows(cursor)` returns the number of rows in the fetched data set, `cursor`.

Examples

There are four rows in the fetched data set, `cursorA`:

```
rowsA = rows(cursorA)

rowsA =

     4
```

To see the four rows of data in `cursorA`, type:

```
cursorA.data

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

See Also

`cols`, `fetch`

Purpose

Set autocommit flag for database or rowlimit for fetch

Syntax

```
set(cursor, 'rowlimit', value)
set(conn, 'autocommit', 'value')
```

Description

`set(cursor, 'rowlimit', value)` sets the `rowlimit` for `fetch` to `value`, where `value` is a positive integer. This is an alternative to setting the `rowlimit` as an argument of `fetch`. You can use `set` to define `rowlimit` for JDBC connections, but not for ODBC connections; for ODBC connections, define `rowlimit` as an argument of `fetch`.

`set(conn, 'autocommit', 'value')` assigns a value to the `autocommit` flag for the database accessed via the connection `conn`, where `value` is `on` or `off`.

- If you set `value` to `on`, the database data is written and committed automatically when you run `insert` or `update`. You cannot use an SQL `rollback` to reverse it and you do not need to use an SQL `commit` because the data is already committed.
- If you set `value` to `off`, the database data is not committed automatically when you run `insert` or `update`. In this case, after you run `insert` or `update`, you can use an SQL `rollback` to reverse the `insert` or `update`. When you are sure the data is correct, follow an `insert` or `update` with an SQL `commit`. This commits the data in the database, after which you cannot use an SQL `rollback` to reverse it.

  Perform an SQL `commit` or `rollback` using `exec`. Note that if you do not run an SQL `commit` after `update` or `insert`, 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.

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

```
help cursor/set.m
help database/set.m
```

Limitations

Microsoft Access does not use SQL commands and therefore has no `rollback` option. Setting the `autocommit` flag for a connection has no effect when you use `insert` or `update` to export data to a Microsoft Access database. The

autocommit flag is always reported as on, and the data is always automatically written and committed.

Examples

Example 1 – Set rowlimit

This example uses set to define the rowlimit. It establishes a JDBC connection, retrieves all data from the emp table, sets the rowlimit to one, and uses fetch with no arguments to retrieve the data. Only one row of data is returned by fetch as you can see from the result of the rows command.

```
conn=database('oracle','scott','tiger',...
     'oracle.jdbc.driver.OracleDriver','jdbc:oracle:oci7:')
curs=exec(conn, 'select * from emp')
set(curs, 'rowlimit', 1)
fetch(curs)
rows(curs)

ans =
     1
```

Run fetch again without a rowlimit argument to retrieve the next row of data.

Example 2 – Set autocommit flag to on

This example shows a database update when the autocommit flag is on. First determine the status of the autocommit flag for the database connection conn1:

```
get(conn1, 'autocommit')

ans =
off
```

The flag is off. Set the flag status to on and verify it:

```
set(conn1, 'autocommit', 'on');
get(conn1, 'autocommit')

ans =
on
```

Insert data, cell array G, into the column names ColNames of the Growth table:

```
insert(conn1, 'Growth', ColNames, G)
```

The data is inserted and committed.

Example 3 – Set autocommit flag to off and commit data

This example shows a database insert when the autocommit flag is off and the data is then committed. First set the autocommit flag to off for database connection connA:

```
set(connA, 'autocommit', 'off');
```

Insert data, cell array A, into the column names ColNames of the Avg_Freight_Cost table:

```
insert(connA, 'Avg_Freight_Cost', ColNames, A)
```

Commit the data:

```
cursorset = exec(connA,'commit')
```

Example 4 – Set autocommit flag to off 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 connB:

```
set(connB, '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 B:

```
update(connB, 'Avg_Freight_Weight', ColNames, B, whereClause)
```

The data was written but not committed.

Roll back the data:

```
cursorset = exec(connB,'rollback')
```

The data in the table is now the same as it was before update was run.

See Also

database, exec, fetch, get, insert, update

update

Purpose

Replace data in database table with data from MATLAB cell array

Syntax

```
update(conn, 'tablename', colnames, data, 'whereclause')
```

Description

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

The status of the `autocommit` flag determines if `update` automatically commits the data or if an SQL `commit` command is needed. View the `autocommit` flag status for the connection using `get` and change it using `set`. Perform an SQL `commit` or `rollback` using `exec`.

To add new rows instead of replacing existing data, use `insert`.

Examples

Example 1 – Update a record

In the `Birthdays` table, update the record where `First_Name` is `Jean`, replacing the current value for `Age` with the new value, `40`. The connection is `conn`.

Define a cell array containing the column name you are updating, `Age`:

```
C = {'Age'}
```

Define a cell array containing the new data:

```
B(1,1) = {40}
```

Perform the update:

```
update(conn, 'Birthdays', C, B, 'where First_Name = ''Jean''')
```

Example 2 – Update followed by SQL 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 `connect`:

```
set(connect, 'autocommit', 'off')
```

Update the data in the column names specified by `C` of the `Error_Rate` table for the record selected by `whereClause` using data contained in the cell array `data`:

```
update(connect, 'Error_Rate', C, data, whereClause)
```

The data was written but not committed.

Roll back the data:

```
cursor_status = exec(connect,'rollback')
```

The `update` was reversed; the data in the table is the same as it was before `update` was run.

See Also

`database`, `insert`, `set`

width

Purpose

Get field size of column in fetched data set

Syntax

```
colsize = width(cursor, column)
```

Description

`colsize = width(cursor, column)` gets the field size of the specified column number, `column`, in the fetched data set, `cursor`.

Examples

Get the width of the first column of the fetched data set, `cursorA`:

```
widthA = width(cursorA, 1)

widthA =

    11
```

The field size of column one is 11 characters (bytes).

See Also

`attr`, `cols`, `columnnames`, `fetch`

Index