Wolfram *Mathematica*® Tutorial Collection

DATABASELINK USER GUIDE

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

©2008 Wolfram Research, Inc.

All rights reserved. No part of this document may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the copyright holder.

Wolfram Research is the holder of the copyright to the Wolfram *Mathematic*a software system ("Software") described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, "look and feel," programming language, and compilation of command names. Use of the Software unless pursuant to the terms of a license granted by Wolfram Research or as otherwise authorized by law is an infringement of the copyright.

Wolfram Research, Inc. and Wolfram Media, Inc. ("Wolfram") make no representations, express, statutory, or implied, with respect to the Software (or any aspect thereof), including, without limitation, any implied warranties of merchantability, interoperability, or fitness for a particular purpose, all of which are expressly disclaimed. Wolfram does not warrant that the functions of the Software will meet your requirements or that the operation of the Software will be uninterrupted or error free. As such, Wolfram does not recommend the use of the software described in this document for applications in which errors or omissions could threaten life, injury or significant loss.

Mathematica, *MathLink*, and *MathSource* are registered trademarks of Wolfram Research, Inc. *J/Link*, *MathLM*, *.NET/Link*, and web*Mathematica* are trademarks of Wolfram Research, Inc. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Macintosh is a registered trademark of Apple Computer, Inc. All other trademarks used herein are the property of their respective owners. *Mathematica* is not associated with Mathematica Policy Research, Inc.

Contents

Introduction to *DatabaseLink* 1
- **Background** 1
- ***Mathematica* Database Applications** 1
- **Features of *DatabaseLink*** 2
- ***DatabaseLink* Technology** 3

Getting Started 3
- **About This Tutorial** 3
- **The Command-Line Interface** 4

Database Connections 11
- **Setting Up a Database** 12
- **Establishing a Connection** 12
- **Connection Information** 14
- **Database Resources** 26
- **Security and Authentication** 31

Descriptive Commands 33
- **Table Structure** 33
- **Column Structure** 37
- **Data Types** 44
- **Schema and Catalogs** 45

Data Commands 47
- **Comparing *Mathematica* and SQL Queries** 47
- ***Mathematica*-Style Queries** 47
- **SQL-Style Queries** 47

The Database Explorer 77
- **The Connection Tool** 78
- **Querying the Database** 78
- **Saving Queries** 80
- **Exporting to *Mathematica*** 81
- **New Connection Wizard** 81

Advanced Topics 83

Data Type Mapping 83

Result Sets 88

Performance 97

Connection Pools 100

Transactions 104

Secure Socket Layer (SSL) 107

Examples 109

Command Cache 109

Graph Database 111

Appendix 114

Database Reference 114

Using the Example Databases 117

Introduction to *DatabaseLink*

Background

Data storage, indexing, and retrieval have long been crucial tasks of many large organizations such as governments, banks, hospitals, and libraries. As human societies have grown increasingly complex, data management requirements have also increased. Some of the new challenges include the complexity of what the data represents, how the data is used, as well as the sheer volume of data. Since the development of modern electronic computers in the latter half of the twentieth century, tools such as relational database management systems (RDBMS) and the Structured Query Language (SQL) have become standards that are widely used for data handling in many different types of organizations.

In a typical organization, many different users need to access the data management system, and hence many database applications are server based. They can be combined with other server-based technologies, often called enterprise technologies, such as web servers, web services, as well as remote computing heterogeneous architectures.

At the current time, there are many different database systems. These range from large-scale, expensive commercial applications that are suitable for high-end uses to freely available open-source tools running on personal computers with operating systems such as Microsoft Windows or Linux.

DatabaseLink is a *Mathematica* application that provides a set of tools allowing convenient integration of *Mathematica* with database management systems.

Mathematica Database Applications

There are a number of important benefits that can be gained from integrating *Mathematica* into a database system.

Mathematica contains a large collection of functions for numerical and symbolic computation that can be applied to data taken from a database. After the computations have been com-

pleted, the results can be stored in a database application, allowing *Mathematica* to work on the results at a later time. *Mathematica* might be used for statistical processing, modeling, or computing some optimal configuration. All of these computations typically require and produce data that can conveniently be stored in a database application.

Database applications can be integrated with many other application types, providing an important form of interoperability. Data derived from one application can be stored in the database. Then, elements of this data can be retrieved by *Mathematica*, used for computation, and the results stored in the database. Finally, another application can extract these results and use them for some further purpose. The central database application is the hub of this computational network; its interaction with *Mathematica* is made possible by *DatabaseLink*.

Features of DatabaseLink

- Connectivity - works with most standard SQL database applications and with databases that are local and network based (including different types of computers).
- The HSQL Database Engine (HSQLDB) - a lightweight database useful for database applications if you don't have an existing database.
- Supplied drivers - built-in support for many important databases, including MySQL, Open Database Connectivity (ODBC), and HSQLDB.
- SQL command interface - you can exploit your knowledge of SQL without learning a new system.
- *Mathematica* command interface - useful if you are familiar with *Mathematica* programming.
- GUI interfaces - the Database Explorer and the Connection Tool provide convenient tools for opening connections and querying the database.
- Access to data and metadata - you can inspect the names of tables and columns, as well as the data in each entry.
- Configurable - common tasks can be simplified and *Mathematica* applications can add their own database information.
- Batch support - provides efficiency when making repeated changes to a database.
- Data type support - works with standard SQL data types, including numbers, strings, binary data, date and time, as well as *Mathematica* expressions.
- Support for advanced features - such as multiple connections and transactions (including rollbacks and savepoints).
- Secure Socket Layer (SSL) support - security for communications with the database.

DatabaseLink Technology

DatabaseLink is based on the commonly used Java database connectivity (JDBC) technology, java.sun.com/products/jdbc/. The package makes extensive use of the *Mathematica* Java toolkit *J/Link* www.wolfram.com/solutions/mathlink/jlink/, though no Java programming is required. The Database Explorer uses the *Mathematica* graphical user interface toolkit *GUIKit*, www.wolfram.com/solutions/guikit.

DatabaseLink comes with a selection of drivers for a number of databases. If it does not include a driver for your database, you can install your own driver, as described in Database Connections: JDBC Connections.

Getting Started

Using This Tutorial

This tutorial contains simple examples of *DatabaseLink* that give an overview of its functionality and some ideas of how to get started. It uses a lightweight database, HSQLDB, that is installed as part of *DatabaseLink*. This allows you to try examples in the documentation without having to install your own database. The other *DatabaseLink* tutorials give detailed reference information.

DatabaseLink provides two styles of interface for working with a database. A command-line interface, which is more flexible and is useful for using database commands inside programs, and a graphical interface, which is simpler to use. Both interfaces are discussed here.

When you have finished trying these examples, you may wish to restore the example database, by using the `DatabaseExamples`` package, as described in "Using the Example Databases".

The Command-Line Interface

Introduction

The command-line interface is a powerful and flexible interface that is particularly appropriate if you want to write programs that use database functionality. This section discusses a number of different operations that use a demonstration database.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

Loading the Package

DatabaseLink is a *Mathematica* add-on application. Before any functions from the package can be used, it must be loaded as follows.

```
In[1]:= Needs["DatabaseLink`"]
```

Connecting to the Database

The details of connecting to a database are described in "Database Connections". The command-line method uses the function `OpenSQLConnection`, which returns a handle that can be used to work with a database. The following opens a connection to an included sample database.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

```
In[2]:= conn = OpenSQLConnection["demo"]
Out[2]= SQLConnection[demo, 7, Open, TransactionIsolationLevel → ReadCommitted]
```

There is also a GUI method to connect to the database that is invoked by executing `OpenSQLConnection` with no arguments. When this is done, the Connection Tool appears; at this point a connection must be opened or the tool canceled before operations can continue.

```
In[3]:= conn1 = OpenSQLConnection[];
```

You can use the Connection Tool to connect to the example database. Further information on how to open a connection to a database is provided in "Database Connections".

Fetching Data

A relational database consists of a set of tables; each table contains data in various categories (typically called columns). Each row of a table contains data values for the different categories. The database application provides functions for managing this data by supporting features such as querying, inserting, updating, or dropping data.

Tables are fundamental to relational databases, and it is important to have a convenient way to list them. You can do this with the `SQLTables` command as follows.

```
In[4]:= SQLTables[conn]

Out[4]= {SQLTable[SAMPLETABLE1, TableType → TABLE]}
```

You can see information on the specific columns in a table with the `SQLColumns` command. An example that provides information on the columns in the *SAMPLETABLE1* table follows.

```
In[5]:= SQLColumns[conn, "SAMPLETABLE1"]

Out[5]= {SQLColumn[{SAMPLETABLE1, ENTRY}, DataTypeName → INTEGER, Nullable → 1, DataLength → Null],
  SQLColumn[{SAMPLETABLE1, VALUE}, DataTypeName → DOUBLE, Nullable → 1, DataLength → Null],
  SQLColumn[{SAMPLETABLE1, NAME}, DataTypeName → VARCHAR, Nullable → 1, DataLength → 2 147 483 647]}
```

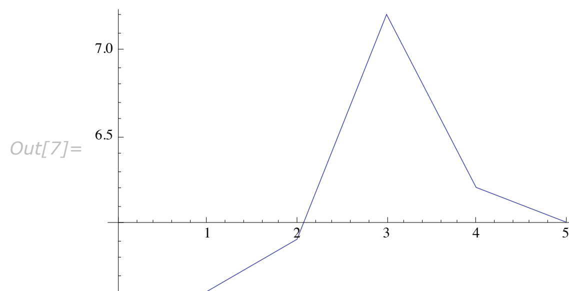
You can retrieve the data in the *SAMPLETABLE1* table by executing an `SQLSelect` command as follows.

```
In[6]:= data = SQLSelect[ conn, "SAMPLETABLE1"]

Out[6]= {{1, 5.6, Day1}, {2, 5.9, Day2}, {3, 7.2, Day3}, {4, 6.2, Day4}, {5, 6., Day5}}
```

The result of the database query is a *Mathematica* list, which can be used in any *Mathematica* command. In the following example the last element of each row is plotted.

```
In[7]:= ListLinePlot[data[[All, 2]]]
```

Out[7]=



The following example retrieves data from the *SALES* table, but adds column headings and outputs the result in a tabular form.

```
In[8]:= SQLSelect[ conn, "SAMPLETABLE1", "ShowColumnHeadings" → True] // TableForm

        ENTRY  VALUE  NAME
        1      5.6    Day1
        2      5.9    Day2
Out[8]= 3      7.2    Day3
        4      6.2    Day4
        5      6.     Day5
```

DatabaseLink also allows you to enter raw SQL commands; this might be useful if you are already familiar with SQL and do not want to learn a new language. Here is an example that shows how to obtain all the data in the *SALES* table.

```
In[9]:= SQLExecute[ conn, "SELECT * FROM SAMPLETABLE1"]
Out[9]= {{1, 5.6, Day1}, {2, 5.9, Day2}, {3, 7.2, Day3}, {4, 6.2, Day4}, {5, 6., Day5}}
```

More information on fetching data is available in "Selecting Data".

Inserting Data

You can use the `SQLInsert` command to insert data in the table. For example, this adds a new row to the *SAMPLETABLE1* table.

```
In[10]:= SQLInsert[ conn, "SAMPLETABLE1", {"ENTRY", "VALUE", "NAME"}, {6, 8.2, "Day6"}]
Out[10]= 1
```

You can see the extra row that has been added.

```
In[11]:= SQLSelect[ conn, "SAMPLETABLE1", "ShowColumnHeadings" → True] // TableForm

         ENTRY  VALUE  NAME
         1      5.6    Day1
         2      5.9    Day2
Out[11]= 3      7.2    Day3
         4      6.2    Day4
         5      6.     Day5
         6      8.2    Day6
```

It is also possible to use a raw SQL command to insert more data. Note how the string being inserted, `Day7`, uses single-quote characters ('). It is also possible to use double-quote characters ("), though these need to be preceded with a *Mathematica* string escape backslash character (\).

```
In[12]:= SQLExecute[conn,
          "INSERT INTO SAMPLETABLE1(ENTRY, VALUE, NAME) VALUES (7, 6.9, 'Day7')"]
Out[12]= 1
```

Another version of raw SQL commands involves using placeholders to represent where the arguments will go and then giving `SQLExecute` a list argument that contains the actual arguments. This is particularly useful since it avoids the need to concatenate strings to form the raw command.

```
In[13]:= SQLExecute[conn,
          "INSERT INTO SAMPLETABLE1(ENTRY, VALUE, NAME) VALUES (`1`, `2`, `3`)",
          {8, 10.5, "Day8"}]
Out[13]= 1
```

This shows the data that is currently in the table.

```
In[14]:= SQLExecute[conn, "SELECT * FROM SAMPLETABLE1"]
Out[14]= {{1, 5.6, Day1}, {2, 5.9, Day2}, {3, 7.2, Day3}, {4, 6.2, Day4},
          {5, 6., Day5}, {6, 8.2, Day6}, {7, 6.9, Day7}, {8, 10.5, Day8}}
```

More information on inserting data is available in "Inserting Data".

Updating Data

You can use the `SQLUpdate` command to update data in the table. Often this is combined with a condition, so that only some of the data is modified. For example, this sets all entries of the *VALUE* column that are greater than 8 to 7.

```
In[15]:= SQLUpdate[conn, "SAMPLETABLE1", {"VALUE"}, {7}, SQLColumn["VALUE"] > 8]
Out[15]= 2
```

You can see the changes that have been made.

```
In[16]:= SQLSelect[ conn, "SAMPLETABLE1", "ShowColumnHeadings" → True] // TableForm

         ENTRY  VALUE  NAME
         1      5.6    Day1
         2      5.9    Day2
         3      7.2    Day3
Out[16]= 4      6.2    Day4
         5      6.     Day5
         7      6.9    Day7
         6      7.     Day6
         8      7.     Day8
```

It is also possible to use a raw SQL command to update data. This sets all rows for which the *VALUE* entry is greater than or equal to 6 to 7.

```
In[17]:= SQLExecute[ conn,
          "UPDATE SAMPLETABLE1 SET VALUE = `1` WHERE VALUE >= `2`", {7, 6}]
Out[17]= 6
```

```
In[18]:= SQLExecute[ conn, "SELECT * FROM SAMPLETABLE1"]
Out[18]= {{1, 5.6, Day1}, {2, 5.9, Day2}, {3, 7., Day3},
          {4, 7., Day4}, {5, 7., Day5}, {7, 7., Day7}, {6, 7., Day6}, {8, 7., Day8}}
```

More information on updating data is available in "Updating Data".

Deleting Data

You can use the `SQLDelete` command to delete data in the table. Often this is combined with a condition, so that only some of the data is modified. For example, this deletes all rows for which the *VALUE* entry is 7 or greater.

```
In[19]:= SQLDelete[ conn, "SAMPLETABLE1", SQLColumn["VALUE"] ≥ 7]
Out[19]= 6
```

You can see the changes that have been made.

```
In[20]:= SQLSelect[ conn, "SAMPLETABLE1", "ShowColumnHeadings" → True] // TableForm

         ENTRY  VALUE  NAME
Out[20]= 1      5.6    Day1
         2      5.9    Day2
```

It is also possible to use a raw SQL command to delete data. This deletes all entries for which the *VALUE* entry is greater than 5.7.

```
In[21]:= SQLExecute[conn, "DELETE FROM SAMPLETABLE1 WHERE VALUE > 5.7"]
Out[21]= 1
```

There is only one row left in the database now.

```
In[22]:= SQLExecute[conn, "SELECT * FROM SAMPLETABLE1"]
Out[22]= {{1, 5.6, Day1}}
```

More information on deleting data is available in "Deleting Data".

Batch Commands

If you want to repeat the same command many times, you can do this by providing repeated arguments in a list. Carrying out the same command like this is much faster than doing each command separately.

The following command inserts two rows.

```
In[23]:= SQLInsert[conn, "SAMPLETABLE1",
          {"ENTRY", "VALUE", "NAME"}, {{2, 5.9, "Day2"}, {3, 7.2, "Day3"}}]
Out[23]= {1, 1}
```

This uses a raw SQL command to insert two more rows.

```
In[24]:= SQLExecute[conn,
          "INSERT INTO SAMPLETABLE1(ENTRY, VALUE, NAME) VALUES (`1`, `2`, `3`)",
          {{4, 6.2, "Day4"}, {5, 6., "Day5"}}]
Out[24]= {1, 1}
```

The result of the insert commands can be seen as follows.

```
In[25]:= SQLExecute[conn, "SELECT * FROM SAMPLETABLE1"]
Out[25]= {{1, 5.6, Day1}, {2, 5.9, Day2}, {3, 7.2, Day3}, {4, 6.2, Day4}, {5, 6., Day5}}
```

Closing the Connection

When you have finished with the connection, you can close it.

```
In[26]:= CloseSQLConnection[conn]
```

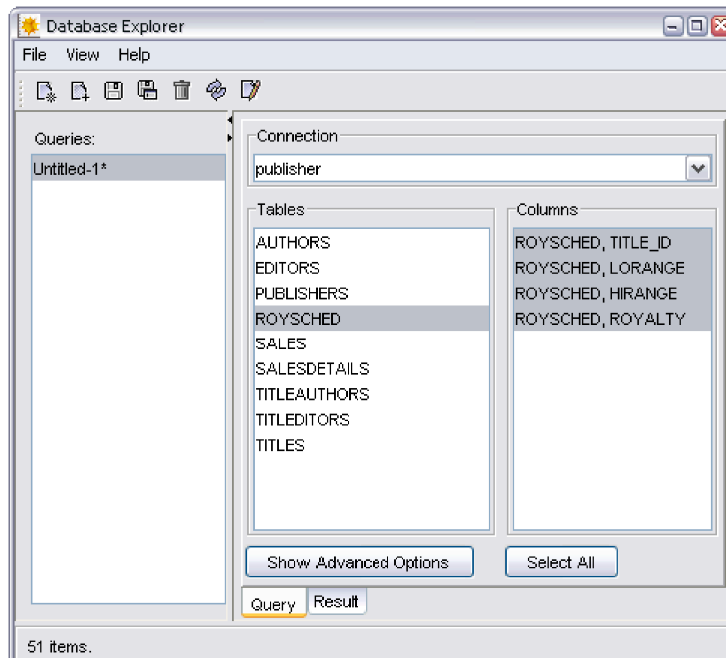
More information on working with connections is provided in "Database Connections". If you have modified the database and want to restore it, you can use the `DatabaseExamples`` package, as described in "Using the Example Databases".

The Database Explorer

The Database Explorer is a graphical interface to database functionality. It can be launched by loading *DatabaseLink* and executing the command `DatabaseExplorer[]`.

```
In[27]:= Needs["DatabaseLink`"];
         DatabaseExplorer[]
Out[28]= - GUIObject -
```

When the Database Explorer opens, you can connect to the different databases that are configured for your system. You can also create new connections. After you connect to a database, you can view the tables and columns, as seen in the following example.



You can then see the data in the database by clicking the Result tab. Here is an example view.

Database Explorer

File View Help

Queries:

Untitled-1*

TITLE_ID	LORANGE	HIRANGE	ROYALTY
BS1011	0	5000	0.1
BS1011	5001	50000	0.12
CP5018	0	2000	0.1
CP5018	2001	4000	0.12
CP5018	4001	50000	0.16
BS1001	0	1000	0.1
BS1001	1001	5000	0.12
BS1001	5001	7000	0.16
BS1001	7001	50000	0.18
PS9999	0	50000	0.1
PY2002	0	1000	0.1
PY2002	1001	5000	0.12
PY2002	5001	50000	0.14
PY2003	0	2000	0.1
PY2003	2001	5000	0.12
PY2003	5001	50000	0.14
UK3004	0	1000	0.1
UK3004	1001	2000	0.12

Query Result

51 items.

The Database Explorer supports many more features, such as forming more complicated queries, saving queries, and creating reports with the result of a query (saved as a *Mathematica* notebook). These are described in "The Database Explorer".

Database Connections

The first step in using a database is making a connection. This part of the tutorial discusses how to do this.

If you are just starting to use *DatabaseLink*, you might want to look at some of the basic examples in this tutorial. Then, to learn if *DatabaseLink* comes with a driver for your database, you might want to study JDBC Connections, which contains further information about adding new drivers. Finally, if you want to give your connection a name, you might want to study Named Connections.

Setting Up a Database

Many users of *DatabaseLink* will have an existing database they wish to connect to and use. If you have one, you should be able to read this documentation and modify it to connect to your own database. If you do not already have a database, you can use HSQLDB (included in *DatabaseLink*). If you want to set up a different type of database, you will need to refer to the specific information for that database. Once you have set up your database, you can continue to use this tutorial to learn how to connect to it.

Establishing a Connection

If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

`OpenSQLConnection[`*name*`]`	connect to a named SQL data source
`OpenSQLConnection[JDBC[`*driver*`,`*url*`]]`	connect to the data source URL using JDBC
`OpenSQLConnection[`*args*`,`*opts*`]`	set options for the connection
`OpenSQLConnection[]`	use the **Connection Tool** to open a connection
`CloseSQLConnection[`*conn*`]`	close a connection
`SQLConnections[]`	list SQL connections
`SQLConnectionInformation[`*conn*`]`	verbose information about an SQL connection

Functions for working with database connections.

This loads *DatabaseLink*.

```
In[29]:= Needs["DatabaseLink`"]
```

Now you can connect to a named database, called *demo*, that is provided by *DatabaseLink* for documentation. Database Resources: Connection Configuration shows how to set up new named connections. You can learn about existing named connections in Named Connections.

`OpenSQLConnection` returns a *Mathematica* expression that refers to the connection. It can be used to make queries on the database.

```
In[30]:= conn = OpenSQLConnection[ "demo"]
Out[30]= SQLConnection[demo, 1, Open, TransactionIsolationLevel → ReadCommitted]
```

`SQLConnections` returns a list of all the open connections.

```
In[31]:= SQLConnections[]
Out[31]= {SQLConnection[demo, 1, Open, TransactionIsolationLevel → ReadCommitted]}
```

In the following example, the tables that are found in the database are returned.

```
In[32]:= SQLTables[conn]
Out[32]= {SQLTable[SAMPLETABLE1, TableType → TABLE]}
```

When you have finished with a connection, you can close it with `CloseSQLConnection`.

```
In[33]:= CloseSQLConnection[conn]

In[34]:= conn
Out[34]= SQLConnection[demo, 1, Closed, <>]
```

There are a number of options that can be given to `OpenSQLConnection`.

option name	*default value*	
"Description"	""	textual description of the connection
"Name"	""	name of the connection
"Username"	""	username to use for connecting
"Password"	""	password to use for connecting
"Catalog"	Automatic	location of the database catalog
"ReadOnly"	Automatic	set the connection to be read only
"TransactionIsolationLevel"	Automatic	set transaction isolation for the connection

`OpenSQLConnection` options.

These options can be used when opening a connection. For instance, the following allows you to use a different username and password for the connection.

```
In[35]:= conn = OpenSQLConnection[ "demo", "Username" -> "sa", "Password" -> ""]
Out[35]= SQLConnection[demo, 2, Open, TransactionIsolationLevel → ReadCommitted]
```

```
In[36]:= CloseSQLConnection[ conn]
```

If you enter `"$Prompt"` as a password, a dialog box opens that will prompt you for the password. This helps keep the password more secure.

```
In[37]:= conn = OpenSQLConnection[ "demo", "Username" -> "sa", "Password" -> "$Prompt"]
Out[37]= SQLConnection[demo, 3, Open, TransactionIsolationLevel → ReadCommitted]
```

Once a connection has been created, certain options can be changed using `SetOptions`.

`"Catalog"`	location of the database catalog
`"ReadOnly"`	whether to open read only
`"TransactionIsolationLevel"`	whether to add transaction isolation

Connection options that can be changed after the connection is created.

This changes the connection to only allow read access to the database.

```
In[38]:= SetOptions[conn, "ReadOnly" → True]
Out[38]= SQLConnection[demo, 3, Open, ReadOnly → True, TransactionIsolationLevel → ReadCommitted]
```

```
In[39]:= CloseSQLConnection[ conn]
```

More information on the `TransactionIsolationLevel` option is found in Transaction Isolation.

Connection Information

Detailed information about a connection can be obtained from `SQLConnectionInformation`. This can be demonstrated in the following sequence.

```
In[40]:= Needs["DatabaseLink`"]
```

This opens a connection to one of the sample databases.

```
In[41]:= conn = OpenSQLConnection[ "demo"]
Out[41]= SQLConnection[demo, 4, Open, TransactionIsolationLevel → ReadCommitted]
```

Here, information on the connection is created.

```
In[42]:= data = SQLConnectionInformation[conn];
```

This prints a tidier form of information on the connection.

```
In[43]:= TableForm[Transpose[data]]
```

```
AllProceduresAreCallable                  True
AllTablesAreSelectable                    True
CatalogSeparator
CatalogTerm
DatabaseMajorVersion                      1
DatabaseMinorVersion                      8
DatabaseProductName                       HSQL Database Engine
DatabaseProductVersion                    1.8.0
DataDefinitionCausesTransactionCommit     True
DataDefinitionIgnoredInTransactions       False
DefaultTransactionIsolationLevel          ReadUncommitted
DeletesAreDetectedForForwardOnly          False
DeletesAreDetectedForScrollInsensitive    False
DeletesAreDetectedForScrollSensitive      False
DoesMaxRowSizeIncludeBlobs                True
DriverMajorVersion                        1
DriverMinorVersion                        8
DriverName                                HSQL Database Engine Driver
DriverVersion                             1.8.0
ExtraNameCharacters
IdentifierQuoteString                     "
InsertsAreDetectedForForwardOnly          False
InsertsAreDetectedForScrollInsensitive    False
InsertsAreDetectedForScrollSensitive      False
IsCatalogAtStartOfTableName               False
JDBCMajorVersion                          3
JDBCMinorVersion                          0
LocatorsUpdateCopy                        False
MaxBinaryLiteralLength                    0
MaxCatalogNameLength                      0
MaxCharLiteralLength                      0
MaxColumnNameLength                       0
MaxColumnsInGroupBy                       0
MaxColumnsInIndex                         0
MaxColumnsInOrderBy                       0
MaxColumnsInSelect                        0
MaxColumnsInTable                         0
MaxConnections                            0
MaxCursorNameLength                       0
MaxIndexLength                            0
MaxProcedureNameLength                    0
MaxRowSize                                0
MaxSchemaNameLength                       0
MaxStatementLength                        0
MaxStatements                             0
MaxTableNameLength                        0
MaxTablesInSelect                         0
```

Out[43]=

MaxUserNameLength	0
NullPlusNonNullIsNull	True
NullsAreSortedAtEnd	False
NullsAreSortedAtStart	False
NullsAreSortedHight	False
NullsAreSortedLow	True
NumericFunctions	ABS,ACOS,ASIN,ATAN,ATAN2,CEILING,COS,COT,DEGREES
OthersDeletesAreVisibleForForwardOnly	False
OthersDeletesAreVisibleForScrollInsensitive	False
OthersDeletesAreVisibleForScrollSensitive	False
OthersInsertsAreVisibleForForwardOnly	False
OthersInsertsAreVisibleForScrollInsensitive	False
OthersInsertsAreVisibleForScrollSensitive	False
OthersUpdatesAreVisibleForForwardOnly	False
OthersUpdatesAreVisibleForScrollInsensitive	False
OthersUpdatesAreVisibleForScrollSensitive	False
OwnDeletesAreVisibleForForwardOnly	False
OwnDeletesAreVisibleForScrollInsensitive	False
OwnDeletesAreVisibleForScrollSensitive	False
OwnInsertsAreVisibleForForwardOnly	False
OwnInsertsAreVisibleForScrollInsensitive	False
OwnInsertsAreVisibleForScrollSensitive	False
OwnUpdatesAreVisibleForForwardOnly	False
OwnUpdatesAreVisibleForScrollInsensitive	False
OwnUpdatesAreVisibleForScrollSensitive	False
ProcedureTerm	
ReadOnly	False
SchemaTerm	SCHEMA
SearchStringEscape	\
SQLKeywords	BEFORE,BIGINT,BINARY,CACHED,DATETIME,LIMIT
SQLStateType	XOpen
StoresLowerCaseIdentifiers	False
StoresLowerCaseQuotedIdentifiers	False
StoresMixedCaseIdentifiers	False
StoresMixedCaseQuotedIdentifiers	False
StoresUpperCaseIdentifiers	True
StoresUpperCaseQuotedIdentifiers	False
StringFunctions	ASCII,BIT_LENGTH,CHAR,CHAR_LENGTH,CHARACTER_LENGTH
SupportsAlterTableWithAddColumn	True
SupportsAlterTableWithDropColumn	True
SupportsANSI92EntryLevelSQL	False
SupportsANSI92FullSQL	False
SupportsANSI92IntermediateSQL	False
SupportsBatchUpdates	True
SupportsCatalogsInDataManipulation	False
SupportsCatalogsInIndexDefinitions	False
SupportsCatalogsInPrivilegeDefinitions	False
SupportsCatalogsInProcedureCalls	False
SupportsCatalogsInTableDefinitions	False
SupportsColumnAliasing	True
SupportsConvert	True
SupportsCoreSQLGrammar	True
SupportsCorrelatedSubqueries	True
SupportsDataDefinitionAndDataManipulationTransactions	False
SupportsDataManipulationTransactionsOnly	True

fferentTableCorrelationNames	True
pressionsInOrderBy	True
tendedSQLGrammar	False
rwardOnlyResultSetReadOnlyConcurrency	True
rwardOnlyResultSetType	True
rwardOnlyResultSetUpdatableConcurrency	False
llOuterJoins	False
tGeneratedKeys	False
oupBy	True
oupByBeyondSelect	True
oupByUnrelated	True
tegrityEnhancementFacility	True
keEscapeClause	True
mitedOuterJoins	True
nimumSQLGrammar	False
xedCaseIdentifiers	False
xedCaseQuotedIdentifiers	True
ltipleOpenResults	False
ltipleResultSets	False
ltipleTransactions	True
medParameters	True
nNullableColumns	True
enCursorsAcrossCommit	False
enCursorsAcrossRollback	False
enStatementsAcrossCommit	True
enStatementsAcrossRollback	True
derByUnrelated	True
terJoins	True
sitionedDelete	False
sitionedUpdate	False
sultSetHoldCursorsOverCommitHoldability	True
sultSetCloseCursorsAtCommitHoldability	False
vepoints	True
hemasInDataManipulation	False
hemasInIndexDefinitions	True
hemasInPrivilegeDefinitions	True
hemasInProcedureCalls	False
hemasInTableDefinitions	True
rollInsensitiveResultSetReadOnlyConcurrency	True
rollInsensitiveResultSetType	True
rollInsensitiveResultSetUpdatableConcurrency	False
rollSensitiveResultSetReadOnlyConcurrency	False
rollSensitiveResultSetType	False
rollSensitiveResultSetUpdatableConcurrency	False
lectForUpdate	False
atementPooling	False
oredProcedures	True
bqueriesInComparisons	True
bqueriesInExists	True
bqueriesInIns	True
bqueriesInQuantifieds	True
bleCorrelationNames	True
adCommitedTransactionIsolationLevel	True
adUncommitedTransactionIsolationLevel	True
peatableReadTransactionIsolationLevel	True

```
SupportsSerializableTransactionIsolationLevel    True
SupportsTransactions                             True
SupportsUnion                                    True
SupportsUnionAll                                 True
SystemFunctions                                  DATABASE,USER,IDENTITY
TimeDateFunctions                                CURDATE,CURTIME,DATEDIFF,DAYNAME,DAY,DAYOFMONTH
UpdatesAreDetectedForForwardOnly                 False
UpdatesAreDetectedForScrollInsensitive           False
UpdatesAreDetectedForScrollSensitive             False
URL                                              jdbc:hsqldb:file:C:\Documents and Settings
UserName                                         SA
UsesLocalFilePerTable                            False
UsesLocalFiles                                   False
```

JDBC Connections

If you do not have a named database connection, you can still connect to the database by using a JDBC setting.

If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

JDBC[*name*,*url*]	a JDBC setting
JDBC[*classname*,*url*]	a JDBC setting that gives the explicit class name for the driver
JDBCDriverNames[]	a list of the names of possible JDBC drivers
JDBCDrivers[]	the details of all JDBC drivers
JDBCDrivers[*name*]	the details of the JDBC driver labeled *name*

This loads the package.

```
In[44]:= Needs["DatabaseLink`"]
```

The following opens a connection to HSQLDB using the file `$UserBaseDirectory/DatabaseResources/Examples/demo`. This works because the package knows what JDBC driver to use for connecting to HSQLDB.

```
In[45]:= conn = OpenSQLConnection[ JDBC["hsqldb",
            ToFileName[{$UserBaseDirectory, "DatabaseResources", "Examples"}, "demo"] ],
              "Name" -> "manualA", "Username" -> "sa"]
Out[45]= SQLConnection[manualA, 5, Open, TransactionIsolationLevel → ReadCommitted]
```

```
In[46]:= CloseSQLConnection[conn]
```

The `JDBCDriverNames` command returns the list of built-in drivers. `hsqldb` appears in this list and therefore you can use the setting `hsqldb` as an argument to `JDBC`.

```
In[47]:= JDBCDriverNames[]

Out[47]= {Microsoft Access(ODBC), hsqldb, HSQL(Memory), HSQL(Server), HSQL(Server+TLS),
  HSQL(Standalone), HSQL(Webserver), HSQL(Webserver+TLS), jtds_sqlserver, jtds_sybase, mysql,
  MySQL(Connector/J), ODBC(DSN), odbc, Oracle(thin), Microsoft SQL Server(jTDS), Sybase(jTDS)}
```

You can get more complete information on all of the built-in drivers by using `JDBCDrivers` without a parameter.

If you want to get information on just one driver, you can do this by giving its name to `JDBCDrivers`. Finding the protocol set for a driver can help to use `OpenSQLConnection`.

```
In[48]:= JDBCDrivers["ODBC(DSN)"]

Out[48]= JDBCDriver[Name → ODBC(DSN), Driver → sun.jdbc.odbc.JdbcOdbcDriver,
  Protocol → jdbc:odbc:, Version → 2., Description →
   JDBC-ODBC Bridge distributed with the Sun JVM.  This driver only works on Windows.,
  Location → C:\Program Files\Wolfram
     Research\Mathematica\7.0\SystemFiles\Links\DatabaseLink\DatabaseResources\odbcdsn.m]
```

The details of how the built-in drivers are configured is described in "Database Resources".

If *DatabaseLink* does not already contain a driver for your database, you can add your own. The driver is a collection of Java classes, and they must be added to *Mathematica* using the standard that *J/Link* provides for adding Java classes. Typically, this is done by adding the class file or a jar file to a Java subdirectory in a *Mathematica* application. One possible location is inside *DatabaseLink* itself. A disadvantage is that if you update *Mathematica*, you may have to copy the new material. Another location would be in an application inside `$UserBaseDirectory` or `$BaseDirectory`; this would not need to be changed if you updated your software.

As an example, you could create an application for connecting to the Oracle database. This could be done by creating an application called `Oracle` inside `$UserBaseDirectory/Applications` or `$BaseDirectory/Applications`. You might have to create some of the directories manually, but you would not need to change anything if you update your software. Another advantage is that you can use the same location to hold a `DatabaseResources` directory, this could hold other configuration information as discussed in "Database Resources".

The following table shows some possible locations that you could use to install drivers for connecting to Oracle.

`$UserBaseDirectory/Applications/Oracle/Java`	location for database driver class files
`$BaseDirectory/Applications/Oracle/Java`	location for database driver class files

Possible locations for database driver class files.

When you have installed the driver classes, you can make a connection. It should be noted that the URL argument you use depends on the server you are using. In the following example, which is not actually configured, a connection is made to an Oracle database using a driver installed in one of the locations previously suggested. The documentation for the JDBC driver will tell you what class and URL to use.

```
In[49]:= OpenSQLConnection[JDBC["oracle.jdbc.driver.OracleDriver",
           "jdbc:oracle:thin:@server.business.com:1999"],
          "Name" → "manualOracle", "Username" → "server1"]
```

This is the most verbose form of `OpenSQLConnection`. Typically, you would want to use information that had been stored previously. This is discussed in "Database Resources".

ODBC Connections

Open Database Connectivity (ODBC) is a general way to connect to SQL databases that is supported in a number of operating systems, particularly Microsoft Windows. *DatabaseLink* comes configured with a driver for ODBC connections. This example, which works only on Windows, shows how to connect to a sample database using ODBC.

Setting Up the Connection

This example uses a sample database file, `publisher.mdb`, which is located inside the *DatabaseLink* package structure. You can find the location by evaluating the following line on your computer.

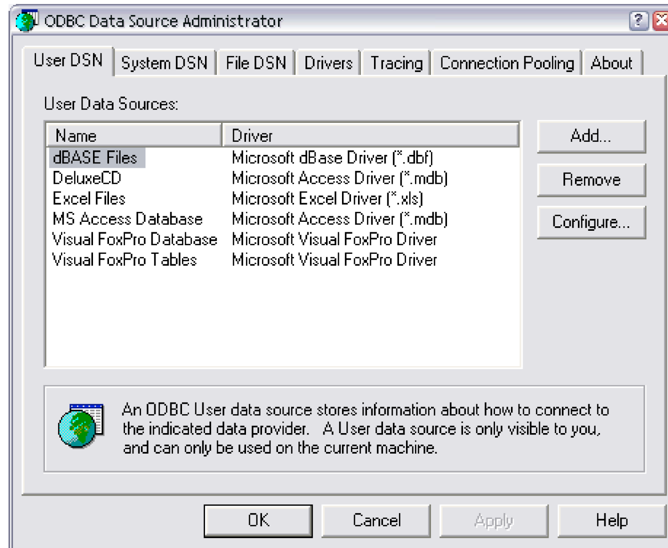
```
In[50]:= Needs["DatabaseLink`"];
         ToFileName[{$DatabaseLinkDirectory}, "Examples"]

Out[51]= C:\Program Files\Wolfram Research\Mathematica\7.0\SystemFiles\Links\DatabaseLink\Examples
```

Typically, it is not a good idea to modify files that are inside of *DatabaseLink*, so you might want to copy it into some other location. One possible location would be inside the `DatabaseResources/Examples` directory inside `$UserBaseDirectory` (it may be necessary to create these directories). You can find the location by evaluating the following on your computer.

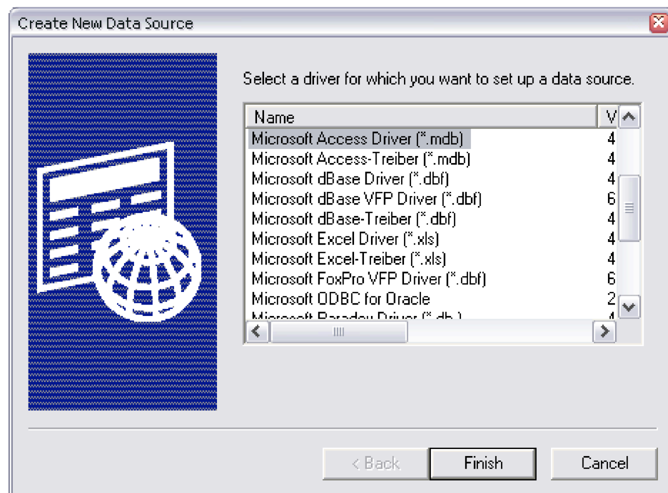
```
In[52]:= ToFileName[{$UserBaseDirectory, "DatabaseResources"}, "Examples"]

Out[52]= C:\Documents and Settings\WRI\Application Data\Mathematica\DatabaseResources\Examples
```

The `publisher.mdb` file is found inside the `Examples` subdirectory.

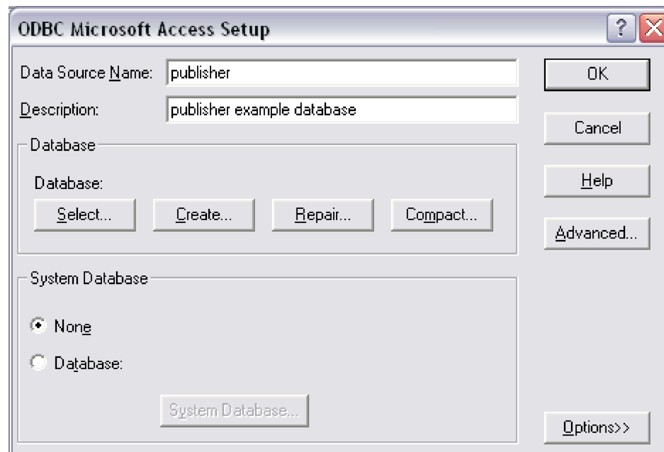
Now, you need to use the ODBC control panel to register the data source. This is typically found in the **Administrative Tools** folder of the Windows **Control Panel**. When it is opened it looks something like the following.



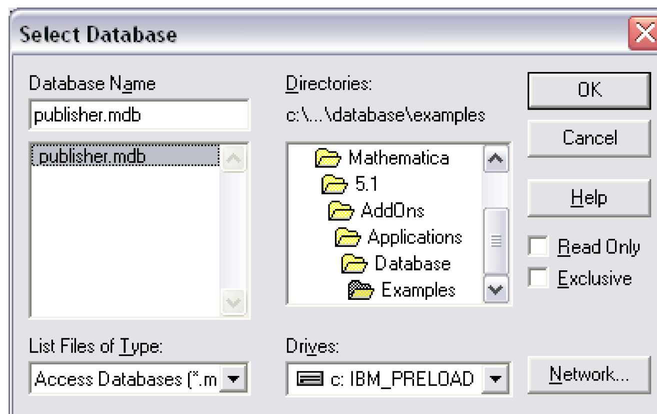
Click the **Add** button, this will bring up the **Create New Data Source** window.



Select **Microsoft Access Driver** and then click **Finish**. This will bring up an **ODBC Microsoft Access Setup** window.



You should fill in the **Data Source Name** text field, using the name "publisher" (this is the name that ODBC will use). Then, click the **Select** button, which allows you to find and select the `publisher.mdb` file.



Click **OK** in each successive window until the control panel has been closed. Note that publisher and its driver now appear in the list of available files in the **ODBC Data Source Administrator** window.

Using the Connection

You should now be able to connect to the ODBC data source that was configured. The following loads *DatabaseLink* and connects to the ODBC data source *publisher*. This will use the ODBC driver that is configured by the package.

```
In[53]:= << DatabaseLink`;
         conn = OpenSQLConnection[JDBC["odbc", "publisher"]];
```

You can use the connection to query the database.

```
In[55]:= SQLTableNames[conn]

Out[55]= {authors, editors, publishers, roysched, sales, salesdetails, titleauthors, titleditors, titles}
```

```
In[56]:= SQLSelect[conn, "publishers", "ShowColumnHeadings" → True] // TableForm

Out[56]= pub_id pub_name                address       city       state
         0736   Second Galaxy Books     100 1st St.   Boston     MA
         0877   Boskone & Helmuth       201 2nd Ave.  Washington DC
         1389   NanoSoft Book Publishers 302 3rd Dr.  Berkeley   CA
```

This closes the connection.

```
In[57]:= CloseSQLConnection[conn]
```

Named Connections

If your work requires that you frequently connect to the same database, it might be beneficial to give this connection a name and use the name in `OpenSQLConnection`. The details of how to set up a named connection are given in "Database Resources". This section describes how to learn what named connections are available.

`DataSourceNames[]`	list of the names of all connections
`DataSources[]`	details of all named connections
`DataSources[`*name*`]`	details of the connection called *name*

Functions for working with named connections.

This loads the package.

```
In[58]:= Needs["DatabaseLink`"]
```

The following lists all the named connections. If you have installed more connections, you may see a larger list.

```
In[59]:= DataSourceNames[]

Out[59]= {demo, graphs, publisher}
```

You can get more complete information on all the connections by using `DataSources`.

```
In[60]:= DataSources[]

Out[60]= {SQLConnection[JDBC[hsqldb, C:\Documents and
              Settings\twj.WRI\Application Data\Mathematica\DatabaseResources\Examples\demo],
           Name → demo, Description → Connection to hsql db for documention., Username → sa,
           Password → , Version → 1.1,
           Location → C:\Documents and Settings\All Users\Application
              Data\Mathematica\Applications\DatabaseLink\DatabaseResources\demo.m],
          SQLConnection[JDBC[hsqldb, C:\Documents and Settings\twj.WRI\Application
              Data\Mathematica\DatabaseResources\Examples\graphs],
           Name → graphs, Description → Connection to the graph database., Username → sa,
           Password → , Version → 1.1,
           Location → C:\Documents and Settings\All Users\Application
              Data\Mathematica\Applications\DatabaseLink\DatabaseResources\graphs.m],
          SQLConnection[JDBC[hsqldb, C:\Documents and Settings\twj.WRI\Application
              Data\Mathematica\DatabaseResources\Examples\publisher],
           Name → publisher, Description → Connection to hsql db for demos.,
           Username → sa,
           Password → , Version → 1.1,
           Location → C:\Documents and Settings\All Users\Application
              Data\Mathematica\Applications\DatabaseLink\DatabaseResources\publisher.m]}
```

You can get information on just one named connection by giving a *name* argument to `DataSources`.

```
In[61]:= DataSources["demo"]

Out[61]= SQLConnection[JDBC[HSQL(Standalone), C:\Documents and
             Settings\brianv\Application Data\Mathematica\DatabaseResources\Examples\demo],
          Name → demo, Description → Connection to HSQL database for documention.,
          Username → sa, Password → , Version → 2.,
          Location → C:\Program Files\Wolfram
             Research\Mathematica\7.0\SystemFiles\Links\DatabaseLink\DatabaseResources\demo.m]
```

Database Timeouts

Database operations typically involve connecting to a server and the possibility of problems accessing the server must be taken into account. Consequently, there is a timeout for database operations such as connecting or executing queries. This timeout is controlled by the global variable `$SQLTimeout`.

option name	*default value*	
$SQLTimeout	Automatic	timeout for making a connection and executing queries

Specification of the timeout for working with the database.

The default value, Automatic, means that the default value given by the driver will be used.

Example Connections

This section shows some sample connection commands and explains how they work.

In this example, you connect to a MySQL database called conn_test running on the computer named databases on port 1234 using the built-in driver with the username test.

```
OpenSQLConnection[ JDBC[ "mysql",
"databases:1234/conn_test"], "Username" -> "test"]
```

In this example, you connect to the same MySQL database as in the previous example, but this time using the driver com.mysql.jdbc.Driver.

```
OpenSQLConnection[ JDBC[ "com.mysql.jdbc.Driver",
"databases:1234/conn_test"], "Username" -> "test"]
```

The first example requires that a JDBC connection mysql has been configured, as described in Database Resources: JDBC Configuration. The second does not require any DatabaseResources configuration. It does require that the JDBC driver, com.mysql.jdbc.Driver, is made available. More information on drivers is found in JDBC Connections.

The Connection Tool

The Connection Tool is a graphical interface tool that simplifies opening a connection to a database. It is launched by executing the command OpenSQLConnection[]. It is described in The Database Explorer: The Connection Tool.

Database Resources

DatabaseLink allows other *Mathematica* applications to hold resource information for database connections in `DatabaseResources` directories. There are a number of possible locations of `DatabaseResources` directories inside `$InstallationDirectory`, `$BaseDirectory`, and `$UserBaseDirectory`.

`DatabaseResourcesPath[]`	`DatabaseResources` directories to search for resources
`$InstallationDirectory/AddOns/ExtraPackages/*`	possible locations for `DatabaseResources` directories
`$InstallationDirectory/AddOns/StandardPackages/*`	
`$InstallationDirectory/AddOns/Autoload/*`	
`$InstallationDirectory/AddOns/Applications/*`	
`$BaseDirectory/Autoload/*`	
`$BaseDirectory/Applications/*`	
`$UserBaseDirectory/Autoload/*`	
`$UserBaseDirectory/Applications/*`	

The command `DatabaseResourcesPath` shows the current locations of `DatabaseResources` directories.

```
In[62]:= Needs["DatabaseLink`"];

In[63]:= DatabaseResourcesPath[]

Out[63]= {C:\Documents and Settings\All Users\Application Data\Mathematica\DatabaseResources\,
 C:\Documents and Settings\WRI\Application Data\Mathematica\DatabaseResources\,
 C:\Documents and Settings\WRI\Application
   Data\Mathematica\Applications\DatabaseLink\DatabaseResources\}
```

`DatabaseResources` directories can hold two sorts of files: those that contain JDBC settings and those that contain connection settings.

JDBC Configuration

Any file that is in a `DatabaseResources` directory with an extension of .m will be inspected to see if it contains possible JDBC configuration information. Here is the format of a JDBC configuration file.

```
JDBCDriver[
    "Name" -> "name",
    "Driver" -> "driverclass",
    "Protocol" -> "protocol",
    "Version" -> 1
]
```

In this format `name` is the name of the connection (as might be used in `OpenSQLConnection`), `driverclass` is the class file of the JDBC driver, and `protocol` is the JDBC protocol. The version of the configuration file is specified by the `Version` setting.

Here is an example file (configured for HSQLDB).

```
JDBCDriver[
    "Name" -> "hsqldb",
    "Driver" -> "org.hsqldb.jdbcDriver",
    "Protocol" -> "jdbc:hsqldb:",
    "Version" -> 1
]
```

This file specifies the driver and protocol to use when `OpenSQLConnection` is invoked for an `hsqldb` connection, such as the following command.

```
OpenSQLConnection[JDBC["hsqldb",ToFileName[{"DatabaseLink","Examples"},"ex
ample"]]
```

Here is another example file (configured for Oracle).

```
JDBCDriver[
    "Name" -> "oracle",
    "Driver" -> "oracle.jdbc.driver.OracleDriver",
    "Protocol" -> "jdbc:oracle:thin:@",
    "Version" -> 1
]
```

This specifies the driver and protocol to use when OpenSQLConnection is invoked for an oracle connection, such as the following command.

```
OpenSQLConnection[JDBC["oracle","server.business.com:1999"],
    "Username" -> "server1"]
```

Note that if you added an application to hold JDBC driver classes (as shown in Database Connections: JDBC Connections), you could create a DatabaseResources directory in the same application to hold JDBC configuration information. The following table shows the layout of an application, named Oracle, that could be used for connecting to the Oracle database.

$UserBaseDirectory/Applications/Oracle/Java	location for database driver class files
$UserBaseDirectory/Applications/Oracle/DatabaseResources	location for JDBC configuration files

When you have installed a new JDBC driver, you might want to confirm that your new driver is accessible to the system. This is described in Database Connections: JDBC Connections.

Connection Configuration

Any file that is in a DatabaseResources directory with an extension of .m will be inspected to see if it contains possible connection configuration information. Here is the format of a connection configuration file.

```
SQLConnection[
  connectdata,
  "Name" -> "name",
  "Description" -> "text",

  "Username" -> "user",
  "Password" -> "pass",
  "RelativePath" -> True|False,
  "Version" -> 1
]
```

Here `connectdata` holds connection data (typically a JDBC setting), `name` is the name of the connection (as might be used in `OpenSQLConnection`), `text` is a textual description of the connection, and `user` and `pass` are the username and password to use when connecting to the database. A password of `$Prompt` causes a GUI to appear to enter the password. If the connection data involves a relative path, this is specified with the `RelativePath` setting. The version of the configuration file is specified by the `Version` setting.

Here is an example file (configured for HSQLDB).

```
SQLConnection[
    JDBC["hsqldb", "../Examples/example"],
    "Name" -> "example",
    "Description" -> "Connection to hsql db for documention.",
    "Username" -> "sa",
    "Password" -> "",
    "RelativePath" -> True,
    "Version" -> 1]
```

This file specifies that HSQLDB should be used to connect to the file `Examples/example`, which is found relative to the location of the configuration file. The username `sa` and a blank password are also given. This connection information is given the name `"example"`. This configuration file supports the following `OpenSQLConnection` command.

```
OpenSQLConnection["example"]
```

Here is another example file (configured for Oracle).

```
SQLConnection[
    JDBC["oracle", "server.business.com:1999"],
    "Name" -> "businessDB",
    "Description" -> "Connection to Oracle db.",
    "Username" -> "server1",
    "Version" -> 1]
```

This specifies connection information to use when `OpenSQLConnection` is invoked with `businessDB`, such as the following command.

```
OpenSQLConnection["businessDB"]
```

Note that if you added an application to hold JDBC driver classes (as shown in Database Connections: JDBC Connections), and JDBC configuration information (as shown previously), you could use the same location for holding the Oracle connection information. The following table shows the layout of an application that could be used for connecting to Oracle.

`$UserBaseDirectory/Applications/Oracle/Java`	location for database driver class files
`$UserBaseDirectory/Applications/Oracle/DatabaseResources`	location for JDBC configuration files
`$UserBaseDirectory/Applications/Oracle/DatabaseResources`	location for connection configuration files

To help you to write the connection configuration file, you can use the command `WriteDataSource`.

```
In[64]:= Needs["DatabaseLink`"];
```

This creates a data source named `testSource`, it will use the `HSQL` database.

```
In[65]:= WriteDataSource["testSource"]

Out[65]= SQLConnection[JDBC[HSQL(Standalone), testSource], Name → testSource,
          Description → , Username → None, Password → None, Properties → {},
          RelativePath → True, UseConnectionPool → Automatic, Catalog → Automatic,
          ReadOnly → Automatic, TransactionIsolationLevel → Automatic, Version → 2.]
```

The new data source shows up in the listing from `DataSourceNames`.

```
In[66]:= DataSourceNames[]

Out[66]= {demo, graphs, publisher, testSource}
```

You can connect to the data source and start to work with it. One benefit of the `HSQL` database is that it will create the database if it does not exist.

```
In[67]:= conn = OpenSQLConnection["testSource"]

Out[67]= SQLConnection[testSource, 2, Open, TransactionIsolationLevel → ReadCommitted]
```

It is typically a good practice to close the connection.

```
In[68]:= CloseSQLConnection[conn]
```

If you want to connect to a database other than `HSQL` you can give a second argument to `WriteDataSource`. For example, the following will write a data source file that uses a `MySQL` database.

```
In[69]:= WriteDataSource["test", "MySQL(Connector/J)", URL → "main/test",
          Username → "user", Password → "password", Location → "User"]

Out[69]= SQLConnection[JDBC[MySQL(Connector/J), main/test], Name → test, Description → , Username → user,
          Password → password, Properties → {}, RelativePath → False, UseConnectionPool → Automatic,
          Catalog → Automatic, ReadOnly → Automatic, TransactionIsolationLevel → Automatic, Version → 2.]
```

Note that this does not communicate with the MySQL server to create the database, `main/test`. It is assumed that the database already exists. However, this is still a convenient way to create a named connection. Note how the parameters such as `Username`, `Password`, and `Location` are set. The choices for `Location` are `"User"` and `"System"`.

If you did not wish to write the connection configuration file yourself, you could use the New Connection Wizard, described in The Database Explorer: New Connection Wizard.

When you have made a new named connection, you might want to confirm that the new connection is accessible to the system. This is described in Database Connections: Named Connections.

Security and Authentication

Many SQL databases can be configured to require a username and password when a connection is made. This is useful for preventing unwanted access and restricting the range of operations that certain users can execute. This attention to security is important since databases are typically server based.

There are a number of issues for *DatabaseLink* that need to be considered when working with passwords. These depend on the level of security you want and how this should be balanced with convenience. Another issue is whether you are running *Mathematica* in a stand-alone mode or inside a server (as in web*Mathematica*).

The most convenient way to work with a password is to place it in a connection configuration file, as described in Database Resources: Connection Configuration. However, the password will be stored in plain text, and an intruder could inspect the configuration file and learn the password. Since this is a security risk, the New Connection Wizard, described in The Database Explorer: New Connection Wizard, does not save a password. However, you can edit the configuration file and add a password. You could provide further protection by ensuring that the permission on the configuration file is restricted to those who are intended to run *Mathematica*.

A higher level of security is obtained if you use a GUI to enter the password, which has the advantage that the password is never stored. The GUI for the password is opened whenever you use a password setting of `"$Prompt"`.

```
In[70]:= conn = OpenSQLConnection["demo", "Username" → "sa", "Password" → "$Prompt"]
```

Here is the dialog box for the password.

You could also enter the password in the `OpenSQLConnection` command, and then make sure that you deleted your *Mathematica* input as soon as you made the connection.

Using a GUI is useful for an interactive session of *Mathematica*, but is not very useful if you run *Mathematica* inside a web server (as in web*Mathematica*). In this case, you have a number of options. You could place the password in a configuration file and use file permissions to restrict access to those who are running the *Mathematica* process in the web server. An alternative would be to store the password in an authenticated mechanism provided by the web server. For example, the Tomcat server provides a mechanism based on JDBC Realms. The database password could be retrieved from the web server and passed to *Mathematica*, which could use it in an `OpenSQLConnection` command. Any hostile inspection of the *Mathematica* code would not find the database password without breaking the web server authentication mechanism.

For greater security, use SSL to protect the transactions between *Mathematica* and the database. This is described in "Secure Socket Layer (SSL)".

Descriptive Commands

Table Structure

Table Description

This section discusses commands that get information about database tables.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

`SQLTableNames[`*conn*`]`	list all table names within a data source
`SQLTableNames[`*conn*`,`*name*`,`*opts*`]`	list all table names that match name within a data source
`SQLTables[`*conn*`]`	list all tables within a data source
`SQLTables[`*conn*`,`*name*`,`*opts*`]`	list all tables that match name within a data source
`SQLTableInformation[`*conn*`]`	list all table information within a data source
`SQLTableInformation[`*conn*`,`*name*`,`*opts*`]`	list all table information for tables that match name within a data source
`SQLTableTypeNames[`*conn*`]`	list the types of table supported in this data source

Functions for retrieving information about tables.

This loads *DatabaseLink* and connects to the *publisher* database.

```
In[71]:= Needs["DatabaseLink`"];
         conn = OpenSQLConnection[ "publisher"];
```

`SQLTableNames` returns a list of the names of the tables within the connection.

```
In[73]:= SQLTableNames[conn]
Out[73]= {AUTHORS, EDITORS, PUBLISHERS, ROYSCHED, SALES, SALESDETAILS, TITLEAUTHORS, TITLEDITORS, TITLES}
```

SQLTables returns a list of SQLTable expressions. These hold information about the tables in a database.

```
In[74]:= SQLTables[conn]

Out[74]= {SQLTable[AUTHORS, TableType → TABLE],
  SQLTable[EDITORS, TableType → TABLE], SQLTable[PUBLISHERS, TableType → TABLE],
  SQLTable[ROYSCHED, TableType → TABLE], SQLTable[SALES, TableType → TABLE],
  SQLTable[SALESDETAILS, TableType → TABLE], SQLTable[TITLEAUTHORS, TableType → TABLE],
  SQLTable[TITLEDITORS, TableType → TABLE], SQLTable[TITLES, TableType → TABLE]}
```

SQLTableInformation returns more complete information about tables.

```
In[75]:= SQLTableInformation[conn] // TableForm

         Null PUBLIC AUTHORS      TABLE Null Null Null Null Null Null MEMORY False
         Null PUBLIC EDITORS      TABLE Null Null Null Null Null Null MEMORY False
         Null PUBLIC PUBLISHERS   TABLE Null Null Null Null Null Null MEMORY False
         Null PUBLIC ROYSCHED     TABLE Null Null Null Null Null Null MEMORY False
Out[75]= Null PUBLIC SALES        TABLE Null Null Null Null Null Null MEMORY False
         Null PUBLIC SALESDETAILS TABLE Null Null Null Null Null Null MEMORY False
         Null PUBLIC TITLEAUTHORS TABLE Null Null Null Null Null Null MEMORY False
         Null PUBLIC TITLEDITORS  TABLE Null Null Null Null Null Null MEMORY False
         Null PUBLIC TITLES       TABLE Null Null Null Null Null Null MEMORY False
```

With each function, you can filter the names of the tables by providing a string to match as the second parameter. An important point is that this filtering is done on the database server, which leads to significant speed enhancements. The following example searches for a table named *AUTHORS*. If no such table existed, the result would be an empty list.

```
In[76]:= SQLTables[conn, "AUTHORS"]

Out[76]= {SQLTable[AUTHORS, TableType → TABLE]}
```

It is also possible to give metacharacters to match more than one table. The metacharacters are '%' which matches zero or more characters, and '_' which matches a single character. The following command returns the names of all tables that start with *TITLE*.

```
In[77]:= SQLTableNames[conn, "TITLE%"]

Out[77]= {TITLEAUTHORS, TITLEDITORS, TITLES}
```

`SQLTables`, `SQLTableNames`, and `SQLTableInformation` take a number of options.

option name	*default value*	
`"TableType"`	`"TABLE"`	type of table to be returned
`"Catalog"`	`None`	database catalog to use
`"Schema"`	`None`	database schema to use
`"ShowColumnHeadings"`	`False`	whether to return headings with the results (`SQLTableInformation` option only)

The option `"TableType"` selects which type of table is returned. Typically, it is the tables of type *TABLE* that are of interest and by default *DatabaseLink* table functions only return information on these. You can use `SQLTableTypeNames` to find all the different types of tables in your data source.

```
In[78]:= SQLTableTypeNames[conn]
Out[78]= {GLOBAL TEMPORARY, SYSTEM TABLE, TABLE, VIEW}
```

If you want to see all the tables in the data source, you can use the result of `SQLTableTypeNames` with the option `"TableType"`. This is demonstrated in the following.

```
In[79]:= SQLTables[conn, "TableType" → SQLTableTypeNames[conn]]
Out[79]= {SQLTable[SYSTEM_ALIASES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_ALLTYPEINFO, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_AUTHORIZATIONS, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_BESTROWIDENTIFIER, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_CACHEINFO, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_CATALOGS, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_CHECK_COLUMN_USAGE, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_CHECK_CONSTRAINTS, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_CHECK_ROUTINE_USAGE, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_CHECK_TABLE_USAGE, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_CLASSPRIVILEGES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_COLLATIONS, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_COLUMNPRIVILEGES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_COLUMNS, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_CROSSREFERENCE, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_INDEXINFO, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_PRIMARYKEYS, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_PROCEDURECOLUMNS, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_PROCEDURES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_PROPERTIES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_ROLE_AUTHORIZATION_DESCRIPTORS, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_SCHEMAS, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_SCHEMATA, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_SEQUENCES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_SESSIONINFO, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_SESSIONS, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_SUPERTABLES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_SUPERTYPES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_TABLEPRIVILEGES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_TABLES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_TABLETYPES, TableType → SYSTEM TABLE],
  SQLTable[SYSTEM_TABLE_CONSTRAINTS, TableType → SYSTEM TABLE],
```

```
SQLTable[SYSTEM_TEXTTABLES, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_TRIGGERCOLUMNS, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_TRIGGERS, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_TYPEINFO, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_UDTATTRIBUTES, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_UDTS, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_USAGE_PRIVILEGES, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_USERS, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_VERSIONCOLUMNS, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_VIEWS, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_VIEW_COLUMN_USAGE, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_VIEW_ROUTINE_USAGE, TableType → SYSTEM TABLE],
SQLTable[SYSTEM_VIEW_TABLE_USAGE, TableType → SYSTEM TABLE],
SQLTable[AUTHORS, TableType → TABLE], SQLTable[EDITORS, TableType → TABLE],
SQLTable[PUBLISHERS, TableType → TABLE], SQLTable[ROYSCHED, TableType → TABLE],
SQLTable[SALES, TableType → TABLE], SQLTable[SALESDETAILS, TableType → TABLE],
SQLTable[TITLEAUTHORS, TableType → TABLE],
SQLTable[TITLEDITORS, TableType → TABLE], SQLTable[TITLES, TableType → TABLE]}
```

The option "ShowColumnHeadings" can be used with SQLTableInformation to return the column headings.

```
In[80]:= SQLTableInformation[conn, "ShowColumnHeadings" → True] // TableForm
```

Out[80]=

TABLE_CAT	TABLE_SCHEM	TABLE_NAME	TABLE_TYPE	REMARKS	TYPE_CAT	TYPE_SCHEM	TYPE_NAME	SELF_REFERENCING
Null	PUBLIC	AUTHORS	TABLE	Null	Null	Null	Null	Null
Null	PUBLIC	EDITORS	TABLE	Null	Null	Null	Null	Null
Null	PUBLIC	PUBLISHERS	TABLE	Null	Null	Null	Null	Null
Null	PUBLIC	ROYSCHED	TABLE	Null	Null	Null	Null	Null
Null	PUBLIC	SALES	TABLE	Null	Null	Null	Null	Null
Null	PUBLIC	SALESDETAILS	TABLE	Null	Null	Null	Null	Null
Null	PUBLIC	TITLEAUTHORS	TABLE	Null	Null	Null	Null	Null
Null	PUBLIC	TITLEDITORS	TABLE	Null	Null	Null	Null	Null
Null	PUBLIC	TITLES	TABLE	Null	Null	Null	Null	Null

This closes the connection.

```
In[81]:= CloseSQLConnection[conn]
```

If the database was designed with particular schema and catalogs, you can also select tables by using the "Catalog" and "Schema" options.

Table Representation

SQLTable expressions hold information about the tables in a database.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the DatabaseExamples` package, as described in "Using the Example Databases".

SQLTable[*table*, *opts*]	expression that represents an SQL table

An example demonstrating `SQLTable` expressions follows. This loads *DatabaseLink* and connects to the *demo* database.

```
In[82]:= Needs["DatabaseLink`"];
         conn = OpenSQLConnection[ "demo"];
```

The `"TableType"` option is used to select the type of the table in the database.

option name	*default value*	
`"TableType"`	`"TABLE"`	type of the table

Now `SQLTables` is used to return a list of the tables in the database; they are returned as `SQLTable` expressions. In this example, a pattern is given to match the names of the tables, and the `"TableType"` option is set to return tables of all types.

```
In[84]:= SQLTables[conn, "%SA%", "TableType" → SQLTableTypeNames[conn]]

Out[84]= {SQLTable[SYSTEM_CHECK_COLUMN_USAGE, TableType → SYSTEM TABLE],
          SQLTable[SYSTEM_CHECK_ROUTINE_USAGE, TableType → SYSTEM TABLE],
          SQLTable[SYSTEM_CHECK_TABLE_USAGE, TableType → SYSTEM TABLE],
          SQLTable[SYSTEM_USAGE_PRIVILEGES, TableType → SYSTEM TABLE],
          SQLTable[SYSTEM_VIEW_COLUMN_USAGE, TableType → SYSTEM TABLE],
          SQLTable[SYSTEM_VIEW_ROUTINE_USAGE, TableType → SYSTEM TABLE],
          SQLTable[SYSTEM_VIEW_TABLE_USAGE, TableType → SYSTEM TABLE],
          SQLTable[SAMPLETABLE1, TableType → TABLE]}
```

This closes the connection.

```
In[85]:= CloseSQLConnection[conn]
```

`SQLTable` expressions can also be used in commands as shown in "Selecting Data".

Column Structure

Column Description

This section discusses commands that get information about database columns.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

SQLColumnNames[*conn*]	list all column names within a data source
SQLColumnNames[*conn*,*name*,*opts*]	list all column names that match *name* within a data source
SQLColumns[*conn*]	list all columns within a data source
SQLColumns[*conn*,*name*,*opts*]	list all columns that match *name* within a data source
SQLColumnInformation[*conn*]	list all column information for tables within a data source
SQLColumnInformation[*conn*,*name*,*opts*]	list all column information for columns that match *name* within a data source

Functions for retrieving information about columns.

This loads *DatabaseLink* and connects to the *demo* database.

```
In[86]:= Needs["DatabaseLink`"];
         conn = OpenSQLConnection[ "demo"];
```

SQLColumnNames returns a list of the column names within a database as a list of pairs of table and column names. For HSQLDB it returns information from many of the *SYSTEM* tables.

```
In[88]:= SQLColumnNames[conn]

Out[88]= {{SYSTEM_ALIASES, OBJECT_TYPE}, {SYSTEM_ALIASES, OBJECT_CAT}, {SYSTEM_ALIASES, OBJECT_SCHEM},
  {SYSTEM_ALIASES, OBJECT_NAME}, {SYSTEM_ALIASES, ALIAS_CAT}, {SYSTEM_ALIASES, ALIAS_SCHEM},
  {SYSTEM_ALIASES, ALIAS}, {SYSTEM_ALLTYPEINFO, TYPE_NAME}, {SYSTEM_ALLTYPEINFO, DATA_TYPE},
  {SYSTEM_ALLTYPEINFO, PRECISION}, {SYSTEM_ALLTYPEINFO, LITERAL_PREFIX},
  {SYSTEM_ALLTYPEINFO, LITERAL_SUFFIX}, {SYSTEM_ALLTYPEINFO, CREATE_PARAMS},
  {SYSTEM_ALLTYPEINFO, NULLABLE}, {SYSTEM_ALLTYPEINFO, CASE_SENSITIVE},
  {SYSTEM_ALLTYPEINFO, SEARCHABLE}, {SYSTEM_ALLTYPEINFO, UNSIGNED_ATTRIBUTE},
  {SYSTEM_ALLTYPEINFO, FIXED_PREC_SCALE}, {SYSTEM_ALLTYPEINFO, AUTO_INCREMENT},
  {SYSTEM_ALLTYPEINFO, LOCAL_TYPE_NAME}, {SYSTEM_ALLTYPEINFO, MINIMUM_SCALE},
  {SYSTEM_ALLTYPEINFO, MAXIMUM_SCALE}, {SYSTEM_ALLTYPEINFO, SQL_DATA_TYPE},
  {SYSTEM_ALLTYPEINFO, SQL_DATETIME_SUB}, {SYSTEM_ALLTYPEINFO, NUM_PREC_RADIX},
  {SYSTEM_ALLTYPEINFO, INTERVAL_PRECISION}, {SYSTEM_ALLTYPEINFO, AS_TAB_COL},
  {SYSTEM_ALLTYPEINFO, AS_PROC_COL}, {SYSTEM_ALLTYPEINFO, MAX_PREC_ACT},
  {SYSTEM_ALLTYPEINFO, MIN_SCALE_ACT}, {SYSTEM_ALLTYPEINFO, MAX_SCALE_ACT},
  {SYSTEM_ALLTYPEINFO, COL_ST_CLS_NAME}, {SYSTEM_ALLTYPEINFO, COL_ST_IS_SUP},
  {SYSTEM_ALLTYPEINFO, STD_MAP_CLS_NAME}, {SYSTEM_ALLTYPEINFO, STD_MAP_IS_SUP},
  {SYSTEM_ALLTYPEINFO, CST_MAP_CLS_NAME}, {SYSTEM_ALLTYPEINFO, CST_MAP_IS_SUP},
  {SYSTEM_ALLTYPEINFO, MCOL_JDBC}, {SYSTEM_ALLTYPEINFO, MCOL_ACT},
  {SYSTEM_ALLTYPEINFO, DEF_OR_FIXED_SCALE}, {SYSTEM_ALLTYPEINFO, REMARKS},
  {SYSTEM_ALLTYPEINFO, TYPE_SUB}, {SYSTEM_AUTHORIZATIONS, AUTHORIZATION_NAME},
  {SYSTEM_AUTHORIZATIONS, AUTHORIZATION_TYPE}, {SYSTEM_BESTROWIDENTIFIER, SCOPE},
  {SYSTEM_BESTROWIDENTIFIER, COLUMN_NAME}, {SYSTEM_BESTROWIDENTIFIER, DATA_TYPE},
  {SYSTEM_BESTROWIDENTIFIER, TYPE_NAME}, {SYSTEM_BESTROWIDENTIFIER, COLUMN_SIZE},
  {SYSTEM_BESTROWIDENTIFIER, BUFFER_LENGTH}, {SYSTEM_BESTROWIDENTIFIER, DECIMAL_DIGITS},
  {SYSTEM_BESTROWIDENTIFIER, PSEUDO_COLUMN}, {SYSTEM_BESTROWIDENTIFIER, TABLE_CAT},
  {SYSTEM_BESTROWIDENTIFIER, TABLE_SCHEM}, {SYSTEM_BESTROWIDENTIFIER, TABLE_NAME},
  {SYSTEM_BESTROWIDENTIFIER, NULLABLE}, {SYSTEM_BESTROWIDENTIFIER, IN_KEY},
  {SYSTEM_CACHEINFO, CACHE_FILE}, {SYSTEM_CACHEINFO, MAX_CACHE_COUNT},
  {SYSTEM_CACHEINFO, MAX_CACHE_BYTES}, {SYSTEM_CACHEINFO, CACHE_SIZE},
  {SYSTEM_CACHEINFO, CACHE_BYTES}, {SYSTEM_CACHEINFO, FILE_FREE_BYTES},
  {SYSTEM_CACHEINFO, FILE_FREE_COUNT}, {SYSTEM_CACHEINFO, FILE_FREE_POS},
  {SYSTEM_CATALOGS, TABLE_CAT}, {SYSTEM_CHECK_COLUMN_USAGE, CONSTRAINT_CATALOG},
  {SYSTEM_CHECK_COLUMN_USAGE, CONSTRAINT_SCHEMA}, {SYSTEM_CHECK_COLUMN_USAGE, CONSTRAINT_NAME},
  {SYSTEM_CHECK_COLUMN_USAGE, TABLE_CATALOG}, {SYSTEM_CHECK_COLUMN_USAGE, TABLE_SCHEMA},
  {SYSTEM_CHECK_COLUMN_USAGE, TABLE_NAME}, {SYSTEM_CHECK_COLUMN_USAGE, COLUMN_NAME},
  {SYSTEM_CHECK_CONSTRAINTS, CONSTRAINT_CATALOG}, {SYSTEM_CHECK_CONSTRAINTS, CONSTRAINT_SCHEMA},
  {SYSTEM_CHECK_CONSTRAINTS, CONSTRAINT_NAME}, {SYSTEM_CHECK_CONSTRAINTS, CHECK_CLAUSE},
  {SYSTEM_CHECK_ROUTINE_USAGE, CONSTRAINT_CATALOG},
  {SYSTEM_CHECK_ROUTINE_USAGE, CONSTRAINT_SCHEMA}, {SYSTEM_CHECK_ROUTINE_USAGE, CONSTRAINT_NAME},
  {SYSTEM_CHECK_ROUTINE_USAGE, SPECIFIC_CATALOG}, {SYSTEM_CHECK_ROUTINE_USAGE, SPECIFIC_SCHEMA},
```

{SYSTEM_CHECK_ROUTINE_USAGE, SPECIFIC_NAME}, {SYSTEM_CHECK_TABLE_USAGE, CONSTRAINT_CATALOG},
{SYSTEM_CHECK_TABLE_USAGE, CONSTRAINT_SCHEMA}, {SYSTEM_CHECK_TABLE_USAGE, CONSTRAINT_NAME},
{SYSTEM_CHECK_TABLE_USAGE, TABLE_CATALOG}, {SYSTEM_CHECK_TABLE_USAGE, TABLE_SCHEMA},
{SYSTEM_CHECK_TABLE_USAGE, TABLE_NAME}, {SYSTEM_CLASSPRIVILEGES, CLASS_CAT},
{SYSTEM_CLASSPRIVILEGES, CLASS_SCHEM}, {SYSTEM_CLASSPRIVILEGES, CLASS_NAME},
{SYSTEM_CLASSPRIVILEGES, GRANTOR}, {SYSTEM_CLASSPRIVILEGES, GRANTEE},
{SYSTEM_CLASSPRIVILEGES, PRIVILEGE}, {SYSTEM_CLASSPRIVILEGES, IS_GRANTABLE},
{SYSTEM_COLLATIONS, COLLATION_CATALOG}, {SYSTEM_COLLATIONS, COLLATION_SCHEMA},
{SYSTEM_COLLATIONS, COLLATION_NAME}, {SYSTEM_COLLATIONS, PAD_ATTRIBUTE},
{SYSTEM_COLLATIONS, COLLATION_TYPE}, {SYSTEM_COLLATIONS, COLLATION_DEFINITION},
{SYSTEM_COLLATIONS, COLLATION_DICTIONARY}, {SYSTEM_COLLATIONS, CHARACTER_REPERTOIRE_NAME},
{SYSTEM_COLUMNPRIVILEGES, TABLE_CAT}, {SYSTEM_COLUMNPRIVILEGES, TABLE_SCHEM},
{SYSTEM_COLUMNPRIVILEGES, TABLE_NAME}, {SYSTEM_COLUMNPRIVILEGES, COLUMN_NAME},
{SYSTEM_COLUMNPRIVILEGES, GRANTOR}, {SYSTEM_COLUMNPRIVILEGES, GRANTEE},
{SYSTEM_COLUMNPRIVILEGES, PRIVILEGE}, {SYSTEM_COLUMNPRIVILEGES, IS_GRANTABLE},
{SYSTEM_COLUMNS, TABLE_CAT}, {SYSTEM_COLUMNS, TABLE_SCHEM}, {SYSTEM_COLUMNS, TABLE_NAME},
{SYSTEM_COLUMNS, COLUMN_NAME}, {SYSTEM_COLUMNS, DATA_TYPE}, {SYSTEM_COLUMNS, TYPE_NAME},
{SYSTEM_COLUMNS, COLUMN_SIZE}, {SYSTEM_COLUMNS, BUFFER_LENGTH},
{SYSTEM_COLUMNS, DECIMAL_DIGITS}, {SYSTEM_COLUMNS, NUM_PREC_RADIX},
{SYSTEM_COLUMNS, NULLABLE}, {SYSTEM_COLUMNS, REMARKS}, {SYSTEM_COLUMNS, COLUMN_DEF},
{SYSTEM_COLUMNS, SQL_DATA_TYPE}, {SYSTEM_COLUMNS, SQL_DATETIME_SUB},
{SYSTEM_COLUMNS, CHAR_OCTET_LENGTH}, {SYSTEM_COLUMNS, ORDINAL_POSITION},
{SYSTEM_COLUMNS, IS_NULLABLE}, {SYSTEM_COLUMNS, SCOPE_CATLOG}, {SYSTEM_COLUMNS, SCOPE_SCHEMA},
{SYSTEM_COLUMNS, SCOPE_TABLE}, {SYSTEM_COLUMNS, SOURCE_DATA_TYPE}, {SYSTEM_COLUMNS, TYPE_SUB},
{SYSTEM_CROSSREFERENCE, PKTABLE_CAT}, {SYSTEM_CROSSREFERENCE, PKTABLE_SCHEM},
{SYSTEM_CROSSREFERENCE, PKTABLE_NAME}, {SYSTEM_CROSSREFERENCE, PKCOLUMN_NAME},
{SYSTEM_CROSSREFERENCE, FKTABLE_CAT}, {SYSTEM_CROSSREFERENCE, FKTABLE_SCHEM},
{SYSTEM_CROSSREFERENCE, FKTABLE_NAME}, {SYSTEM_CROSSREFERENCE, FKCOLUMN_NAME},
{SYSTEM_CROSSREFERENCE, KEY_SEQ}, {SYSTEM_CROSSREFERENCE, UPDATE_RULE},
{SYSTEM_CROSSREFERENCE, DELETE_RULE}, {SYSTEM_CROSSREFERENCE, FK_NAME},
{SYSTEM_CROSSREFERENCE, PK_NAME}, {SYSTEM_CROSSREFERENCE, DEFERRABILITY},
{SYSTEM_INDEXINFO, TABLE_CAT}, {SYSTEM_INDEXINFO, TABLE_SCHEM},
{SYSTEM_INDEXINFO, TABLE_NAME}, {SYSTEM_INDEXINFO, NON_UNIQUE},
{SYSTEM_INDEXINFO, INDEX_QUALIFIER}, {SYSTEM_INDEXINFO, INDEX_NAME}, {SYSTEM_INDEXINFO, TYPE},
{SYSTEM_INDEXINFO, ORDINAL_POSITION}, {SYSTEM_INDEXINFO, COLUMN_NAME},
{SYSTEM_INDEXINFO, ASC_OR_DESC}, {SYSTEM_INDEXINFO, CARDINALITY}, {SYSTEM_INDEXINFO, PAGES},
{SYSTEM_INDEXINFO, FILTER_CONDITION}, {SYSTEM_PRIMARYKEYS, TABLE_CAT},
{SYSTEM_PRIMARYKEYS, TABLE_SCHEM}, {SYSTEM_PRIMARYKEYS, TABLE_NAME},
{SYSTEM_PRIMARYKEYS, COLUMN_NAME}, {SYSTEM_PRIMARYKEYS, KEY_SEQ},
{SYSTEM_PRIMARYKEYS, PK_NAME}, {SYSTEM_PROCEDURECOLUMNS, PROCEDURE_CAT},
{SYSTEM_PROCEDURECOLUMNS, PROCEDURE_SCHEM}, {SYSTEM_PROCEDURECOLUMNS, PROCEDURE_NAME},
{SYSTEM_PROCEDURECOLUMNS, COLUMN_NAME}, {SYSTEM_PROCEDURECOLUMNS, COLUMN_TYPE},
{SYSTEM_PROCEDURECOLUMNS, DATA_TYPE}, {SYSTEM_PROCEDURECOLUMNS, TYPE_NAME},
{SYSTEM_PROCEDURECOLUMNS, PRECISION}, {SYSTEM_PROCEDURECOLUMNS, LENGTH},
{SYSTEM_PROCEDURECOLUMNS, SCALE}, {SYSTEM_PROCEDURECOLUMNS, RADIX},
{SYSTEM_PROCEDURECOLUMNS, NULLABLE}, {SYSTEM_PROCEDURECOLUMNS, REMARKS},
{SYSTEM_PROCEDURECOLUMNS, SPECIFIC_NAME}, {SYSTEM_PROCEDURECOLUMNS, SEQ},
{SYSTEM_PROCEDURES, PROCEDURE_CAT}, {SYSTEM_PROCEDURES, PROCEDURE_SCHEM},
{SYSTEM_PROCEDURES, PROCEDURE_NAME}, {SYSTEM_PROCEDURES, NUM_INPUT_PARAMS},
{SYSTEM_PROCEDURES, NUM_OUTPUT_PARAMS}, {SYSTEM_PROCEDURES, NUM_RESULT_SETS},
{SYSTEM_PROCEDURES, REMARKS}, {SYSTEM_PROCEDURES, PROCEDURE_TYPE},
{SYSTEM_PROCEDURES, ORIGIN}, {SYSTEM_PROCEDURES, SPECIFIC_NAME},
{SYSTEM_PROPERTIES, PROPERTY_SCOPE}, {SYSTEM_PROPERTIES, PROPERTY_NAMESPACE},
{SYSTEM_PROPERTIES, PROPERTY_NAME}, {SYSTEM_PROPERTIES, PROPERTY_VALUE},
{SYSTEM_PROPERTIES, PROPERTY_CLASS}, {SYSTEM_ROLE_AUTHORIZATION_DESCRIPTORS, ROLE_NAME},
{SYSTEM_ROLE_AUTHORIZATION_DESCRIPTORS, GRANTEE},
{SYSTEM_ROLE_AUTHORIZATION_DESCRIPTORS, GRANTOR},
{SYSTEM_ROLE_AUTHORIZATION_DESCRIPTORS, IS_GRANTABLE}, {SYSTEM_SCHEMAS, TABLE_SCHEM},
{SYSTEM_SCHEMAS, TABLE_CATALOG}, {SYSTEM_SCHEMAS, IS_DEFAULT},
{SYSTEM_SCHEMATA, CATALOG_NAME}, {SYSTEM_SCHEMATA, SCHEMA_NAME},
{SYSTEM_SCHEMATA, SCHEMA_OWNER}, {SYSTEM_SCHEMATA, DEFAULT_CHARACTER_SET_CATALOG},
{SYSTEM_SCHEMATA, DEFAULT_CHARACTER_SET_SCHEMA}, {SYSTEM_SCHEMATA, DEFAULT_CHARACTER_SET_NAME},
{SYSTEM_SCHEMATA, SQL_PATH}, {SYSTEM_SEQUENCES, SEQUENCE_CATALOG},
{SYSTEM_SEQUENCES, SEQUENCE_SCHEMA}, {SYSTEM_SEQUENCES, SEQUENCE_NAME},
{SYSTEM_SEQUENCES, DTD_IDENTIFIER}, {SYSTEM_SEQUENCES, MAXIMUM_VALUE},
{SYSTEM_SEQUENCES, MINIMUM_VALUE}, {SYSTEM_SEQUENCES, INCREMENT},
{SYSTEM_SEQUENCES, CYCLE_OPTION}, {SYSTEM_SEQUENCES, START_WITH},
{SYSTEM_SESSIONINFO, KEY}, {SYSTEM_SESSIONINFO, VALUE}, {SYSTEM_SESSIONS, SESSION_ID},
{SYSTEM_SESSIONS, CONNECTED}, {SYSTEM_SESSIONS, USER_NAME}, {SYSTEM_SESSIONS, IS_ADMIN},
{SYSTEM_SESSIONS, AUTOCOMMIT}, {SYSTEM_SESSIONS, READONLY}, {SYSTEM_SESSIONS, MAXROWS},
{SYSTEM_SESSIONS, LAST_IDENTITY}, {SYSTEM_SESSIONS, TRANSACTION_SIZE},
{SYSTEM_SESSIONS, SCHEMA}, {SYSTEM_SUPERTABLES, TABLE_CAT}, {SYSTEM_SUPERTABLES, TABLE_SCHEM},

```
{SYSTEM_SUPERTABLES, TABLE_NAME}, {SYSTEM_SUPERTABLES, SUPERTABLE_NAME},
{SYSTEM_SUPERTYPES, TYPE_CAT}, {SYSTEM_SUPERTYPES, TYPE_SCHEM}, {SYSTEM_SUPERTYPES, TYPE_NAME},
{SYSTEM_SUPERTYPES, SUPERTYPE_CAT}, {SYSTEM_SUPERTYPES, SUPERTYPE_SCHEM},
{SYSTEM_SUPERTYPES, SUPERTYPE_NAME}, {SYSTEM_TABLEPRIVILEGES, TABLE_CAT},
{SYSTEM_TABLEPRIVILEGES, TABLE_SCHEM}, {SYSTEM_TABLEPRIVILEGES, TABLE_NAME},
{SYSTEM_TABLEPRIVILEGES, GRANTOR}, {SYSTEM_TABLEPRIVILEGES, GRANTEE},
{SYSTEM_TABLEPRIVILEGES, PRIVILEGE}, {SYSTEM_TABLEPRIVILEGES, IS_GRANTABLE},
{SYSTEM_TABLES, TABLE_CAT}, {SYSTEM_TABLES, TABLE_SCHEM},
{SYSTEM_TABLES, TABLE_NAME}, {SYSTEM_TABLES, TABLE_TYPE}, {SYSTEM_TABLES, REMARKS},
{SYSTEM_TABLES, TYPE_CAT}, {SYSTEM_TABLES, TYPE_SCHEM}, {SYSTEM_TABLES, TYPE_NAME},
{SYSTEM_TABLES, SELF_REFERENCING_COL_NAME}, {SYSTEM_TABLES, REF_GENERATION},
{SYSTEM_TABLES, HSQLDB_TYPE}, {SYSTEM_TABLES, READ_ONLY}, {SYSTEM_TABLETYPES, TABLE_TYPE},
{SYSTEM_TABLE_CONSTRAINTS, CONSTRAINT_CATALOG}, {SYSTEM_TABLE_CONSTRAINTS, CONSTRAINT_SCHEMA},
{SYSTEM_TABLE_CONSTRAINTS, CONSTRAINT_NAME}, {SYSTEM_TABLE_CONSTRAINTS, CONSTRAINT_TYPE},
{SYSTEM_TABLE_CONSTRAINTS, TABLE_CATALOG}, {SYSTEM_TABLE_CONSTRAINTS, TABLE_SCHEMA},
{SYSTEM_TABLE_CONSTRAINTS, TABLE_NAME}, {SYSTEM_TABLE_CONSTRAINTS, IS_DEFERRABLE},
{SYSTEM_TABLE_CONSTRAINTS, INITIALLY_DEFERRED}, {SYSTEM_TEXTTABLES, TABLE_CAT},
{SYSTEM_TEXTTABLES, TABLE_SCHEM}, {SYSTEM_TEXTTABLES, TABLE_NAME},
{SYSTEM_TEXTTABLES, DATA_SOURCE_DEFINTION}, {SYSTEM_TEXTTABLES, FILE_PATH},
{SYSTEM_TEXTTABLES, FILE_ENCODING}, {SYSTEM_TEXTTABLES, FIELD_SEPARATOR},
{SYSTEM_TEXTTABLES, VARCHAR_SEPARATOR}, {SYSTEM_TEXTTABLES, LONGVARCHAR_SEPARATOR},
{SYSTEM_TEXTTABLES, IS_IGNORE_FIRST}, {SYSTEM_TEXTTABLES, IS_ALL_QUOTED},
{SYSTEM_TEXTTABLES, IS_QUOTED}, {SYSTEM_TEXTTABLES, IS_DESC},
{SYSTEM_TRIGGERCOLUMNS, TRIGGER_CAT}, {SYSTEM_TRIGGERCOLUMNS, TRIGGER_SCHEM},
{SYSTEM_TRIGGERCOLUMNS, TRIGGER_NAME}, {SYSTEM_TRIGGERCOLUMNS, TABLE_CAT},
{SYSTEM_TRIGGERCOLUMNS, TABLE_SCHEM}, {SYSTEM_TRIGGERCOLUMNS, TABLE_NAME},
{SYSTEM_TRIGGERCOLUMNS, COLUMN_NAME}, {SYSTEM_TRIGGERCOLUMNS, COLUMN_LIST},
{SYSTEM_TRIGGERCOLUMNS, COLUMN_USAGE}, {SYSTEM_TRIGGERS, TRIGGER_CAT},
{SYSTEM_TRIGGERS, TRIGGER_SCHEM}, {SYSTEM_TRIGGERS, TRIGGER_NAME},
{SYSTEM_TRIGGERS, TRIGGER_TYPE}, {SYSTEM_TRIGGERS, TRIGGERING_EVENT},
{SYSTEM_TRIGGERS, TABLE_CAT}, {SYSTEM_TRIGGERS, TABLE_SCHEM},
{SYSTEM_TRIGGERS, BASE_OBJECT_TYPE}, {SYSTEM_TRIGGERS, TABLE_NAME},
{SYSTEM_TRIGGERS, COLUMN_NAME}, {SYSTEM_TRIGGERS, REFERENCING_NAMES},
{SYSTEM_TRIGGERS, WHEN_CLAUSE}, {SYSTEM_TRIGGERS, STATUS}, {SYSTEM_TRIGGERS, DESCRIPTION},
{SYSTEM_TRIGGERS, ACTION_TYPE}, {SYSTEM_TRIGGERS, TRIGGER_BODY}, {SYSTEM_TYPEINFO, TYPE_NAME},
{SYSTEM_TYPEINFO, DATA_TYPE}, {SYSTEM_TYPEINFO, PRECISION}, {SYSTEM_TYPEINFO, LITERAL_PREFIX},
{SYSTEM_TYPEINFO, LITERAL_SUFFIX}, {SYSTEM_TYPEINFO, CREATE_PARAMS},
{SYSTEM_TYPEINFO, NULLABLE}, {SYSTEM_TYPEINFO, CASE_SENSITIVE}, {SYSTEM_TYPEINFO, SEARCHABLE},
{SYSTEM_TYPEINFO, UNSIGNED_ATTRIBUTE}, {SYSTEM_TYPEINFO, FIXED_PREC_SCALE},
{SYSTEM_TYPEINFO, AUTO_INCREMENT}, {SYSTEM_TYPEINFO, LOCAL_TYPE_NAME},
{SYSTEM_TYPEINFO, MINIMUM_SCALE}, {SYSTEM_TYPEINFO, MAXIMUM_SCALE},
{SYSTEM_TYPEINFO, SQL_DATA_TYPE}, {SYSTEM_TYPEINFO, SQL_DATETIME_SUB},
{SYSTEM_TYPEINFO, NUM_PREC_RADIX}, {SYSTEM_TYPEINFO, TYPE_SUB},
{SYSTEM_UDTATTRIBUTES, TYPE_CAT}, {SYSTEM_UDTATTRIBUTES, TYPE_SCHEM},
{SYSTEM_UDTATTRIBUTES, TYPE_NAME}, {SYSTEM_UDTATTRIBUTES, ATTR_NAME},
{SYSTEM_UDTATTRIBUTES, DATA_TYPE}, {SYSTEM_UDTATTRIBUTES, ATTR_TYPE_NAME},
{SYSTEM_UDTATTRIBUTES, ATTR_SIZE}, {SYSTEM_UDTATTRIBUTES, DECIMAL_DIGITS},
{SYSTEM_UDTATTRIBUTES, NUM_PREC_RADIX}, {SYSTEM_UDTATTRIBUTES, NULLABLE},
{SYSTEM_UDTATTRIBUTES, REMARKS}, {SYSTEM_UDTATTRIBUTES, ATTR_DEF},
{SYSTEM_UDTATTRIBUTES, SQL_DATA_TYPE}, {SYSTEM_UDTATTRIBUTES, SQL_DATETIME_SUB},
{SYSTEM_UDTATTRIBUTES, CHAR_OCTET_LENGTH}, {SYSTEM_UDTATTRIBUTES, ORDINAL_POSITION},
{SYSTEM_UDTATTRIBUTES, IS_NULLABLE}, {SYSTEM_UDTATTRIBUTES, SCOPE_CATALOG},
{SYSTEM_UDTATTRIBUTES, SCOPE_SCHEMA}, {SYSTEM_UDTATTRIBUTES, SCOPE_TABLE},
{SYSTEM_UDTATTRIBUTES, SOURCE_DATA_TYPE}, {SYSTEM_UDTS, TYPE_CAT}, {SYSTEM_UDTS, TYPE_SCHEM},
{SYSTEM_UDTS, TYPE_NAME}, {SYSTEM_UDTS, CLASS_NAME}, {SYSTEM_UDTS, DATA_TYPE},
{SYSTEM_UDTS, REMARKS}, {SYSTEM_UDTS, BASE_TYPE}, {SYSTEM_USAGE_PRIVILEGES, GRANTOR},
{SYSTEM_USAGE_PRIVILEGES, GRANTEE}, {SYSTEM_USAGE_PRIVILEGES, OBJECT_CATALOG},
{SYSTEM_USAGE_PRIVILEGES, OBJECT_SCHEMA}, {SYSTEM_USAGE_PRIVILEGES, OBJECT_NAME},
{SYSTEM_USAGE_PRIVILEGES, OBJECT_TYPE}, {SYSTEM_USAGE_PRIVILEGES, IS_GRANTABLE},
{SYSTEM_USERS, USER}, {SYSTEM_USERS, ADMIN}, {SYSTEM_VERSIONCOLUMNS, SCOPE},
{SYSTEM_VERSIONCOLUMNS, COLUMN_NAME}, {SYSTEM_VERSIONCOLUMNS, DATA_TYPE},
{SYSTEM_VERSIONCOLUMNS, TYPE_NAME}, {SYSTEM_VERSIONCOLUMNS, COLUMN_SIZE},
{SYSTEM_VERSIONCOLUMNS, BUFFER_LENGTH}, {SYSTEM_VERSIONCOLUMNS, DECIMAL_DIGITS},
{SYSTEM_VERSIONCOLUMNS, PSEUDO_COLUMN}, {SYSTEM_VERSIONCOLUMNS, TABLE_CAT},
{SYSTEM_VERSIONCOLUMNS, TABLE_SCHEM}, {SYSTEM_VERSIONCOLUMNS, TABLE_NAME},
{SYSTEM_VIEWS, TABLE_CATALOG}, {SYSTEM_VIEWS, TABLE_SCHEMA}, {SYSTEM_VIEWS, TABLE_NAME},
{SYSTEM_VIEWS, VIEW_DEFINITION}, {SYSTEM_VIEWS, CHECK_OPTION}, {SYSTEM_VIEWS, IS_UPDATABLE},
{SYSTEM_VIEWS, VALID}, {SYSTEM_VIEW_COLUMN_USAGE, VIEW_CATALOG},
{SYSTEM_VIEW_COLUMN_USAGE, VIEW_SCHEMA}, {SYSTEM_VIEW_COLUMN_USAGE, VIEW_NAME},
{SYSTEM_VIEW_COLUMN_USAGE, TABLE_CATALOG}, {SYSTEM_VIEW_COLUMN_USAGE, TABLE_SCHEMA},
{SYSTEM_VIEW_COLUMN_USAGE, TABLE_NAME}, {SYSTEM_VIEW_COLUMN_USAGE, COLUMN_NAME},
{SYSTEM_VIEW_ROUTINE_USAGE, TABLE_CATALOG}, {SYSTEM_VIEW_ROUTINE_USAGE, TABLE_SCHEMA},
```

```
{SYSTEM_VIEW_ROUTINE_USAGE, TABLE_NAME}, {SYSTEM_VIEW_ROUTINE_USAGE, SPECIFIC_CATALOG},
{SYSTEM_VIEW_ROUTINE_USAGE, SPECIFIC_SCHEMA}, {SYSTEM_VIEW_ROUTINE_USAGE, SPECIFIC_NAME},
{SYSTEM_VIEW_TABLE_USAGE, VIEW_CATALOG}, {SYSTEM_VIEW_TABLE_USAGE, VIEW_SCHEMA},
{SYSTEM_VIEW_TABLE_USAGE, VIEW_NAME}, {SYSTEM_VIEW_TABLE_USAGE, TABLE_CATALOG},
{SYSTEM_VIEW_TABLE_USAGE, TABLE_SCHEMA}, {SYSTEM_VIEW_TABLE_USAGE, TABLE_NAME},
{SAMPLETABLE1, ENTRY}, {SAMPLETABLE1, VALUE}, {SAMPLETABLE1, NAME}}
```

It is possible to use metacharacters that will match names. The metacharacters are '%' for zero or more characters and '_' for a single character. The following command matches columns in tables that have names starting with "SA".

```
In[89]:= SQLColumnNames[conn, "SA%"]

Out[89]= {{SAMPLETABLE1, ENTRY}, {SAMPLETABLE1, VALUE}, {SAMPLETABLE1, NAME}}
```

`SQLColumns` returns a list of `SQLColumn` expressions. `SQLColumn` expressions are sometimes useful for structural arguments in database commands, as described in Argument Sequences in SQL-Style Queries, because they contain information on the table name, column name, data type, whether an entry can be set to `Null`, and the data length.

```
In[90]:= SQLColumns[conn, "SA%"]

Out[90]= {SQLColumn[{SAMPLETABLE1, ENTRY}, DataTypeName → INTEGER, Nullable → 1, DataLength → Null],
 SQLColumn[{SAMPLETABLE1, VALUE}, DataTypeName → DOUBLE, Nullable → 1, DataLength → Null],
 SQLColumn[{SAMPLETABLE1, NAME}, DataTypeName → VARCHAR, Nullable → 1, DataLength → 2 147 483 647]}
```

`SQLColumnInformation` returns more information about the columns.

```
In[91]:= SQLColumnInformation[conn, "SA%"] // TableForm

         Null  PUBLIC  SAMPLETABLE1  ENTRY  4   INTEGER  Null           4     Null  10    1  Null  Null  4   Null  Null
Out[91]= Null  PUBLIC  SAMPLETABLE1  VALUE  8   DOUBLE   Null           8     Null  10    1  Null  Null  8   Null  Null
         Null  PUBLIC  SAMPLETABLE1  NAME   12  VARCHAR  2 147 483 647  Null  Null  Null  1  Null  Null  12  Null  Null
```

You can filter the names of the columns by providing a list of metacharacters to match the table and column names. The following command searches in all tables to return all columns that start with V.

```
In[92]:= SQLColumnNames[conn, {"%", "V%"}]

Out[92]= {{SYSTEM_VIEWS, VALID}, {SYSTEM_SESSIONINFO, VALUE}, {SAMPLETABLE1, VALUE},
 {SYSTEM_TEXTTABLES, VARCHAR_SEPARATOR}, {SYSTEM_VIEW_COLUMN_USAGE, VIEW_CATALOG},
 {SYSTEM_VIEW_TABLE_USAGE, VIEW_CATALOG}, {SYSTEM_VIEWS, VIEW_DEFINITION},
 {SYSTEM_VIEW_COLUMN_USAGE, VIEW_NAME}, {SYSTEM_VIEW_TABLE_USAGE, VIEW_NAME},
 {SYSTEM_VIEW_COLUMN_USAGE, VIEW_SCHEMA}, {SYSTEM_VIEW_TABLE_USAGE, VIEW_SCHEMA}}
```

You can find all the columns in a single table by specifying the table name.

```
In[93]:= SQLColumnNames[conn, {"SAMPLETABLE1", "%"}]

Out[93]= {{SAMPLETABLE1, ENTRY}, {SAMPLETABLE1, VALUE}, {SAMPLETABLE1, NAME}}
```

You can also give a `SQLTable` argument.

```
In[94]:= SQLColumnNames[conn, SQLTable["SAMPLETABLE1"]]
Out[94]= {{SAMPLETABLE1, ENTRY}, {SAMPLETABLE1, VALUE}, {SAMPLETABLE1, NAME}}
```

`SQLColumnNames` returns a list where each entry is a list of the table name and the column names. If you want a list of just the column names, you can use *Mathematica* part notation, entered with `[[All, 2]]`, to extract just the second elements.

```
In[95]:= SQLColumnNames[conn, SQLTable["SAMPLETABLE1"]][[All, 2]]
Out[95]= {ENTRY, VALUE, NAME}
```

In addition, you can give an `SQLColumn` argument.

```
In[96]:= SQLColumnNames[conn, SQLColumn["V%"]]
Out[96]= {{SYSTEM_VIEWS, VALID}, {SYSTEM_SESSIONINFO, VALUE}, {SAMPLETABLE1, VALUE},
  {SYSTEM_TEXTTABLES, VARCHAR_SEPARATOR}, {SYSTEM_VIEW_COLUMN_USAGE, VIEW_CATALOG},
  {SYSTEM_VIEW_TABLE_USAGE, VIEW_CATALOG}, {SYSTEM_VIEWS, VIEW_DEFINITION},
  {SYSTEM_VIEW_COLUMN_USAGE, VIEW_NAME}, {SYSTEM_VIEW_TABLE_USAGE, VIEW_NAME},
  {SYSTEM_VIEW_COLUMN_USAGE, VIEW_SCHEMA}, {SYSTEM_VIEW_TABLE_USAGE, VIEW_SCHEMA}}
```

`SQLColumns`, `SQLColumnNames`, and `SQLColumnInformation` take a number of options.

option name	*default value*	
"Catalog"	None	database catalog to use
"Schema"	None	database schema to use
"ShowColumnHeadings"	False	whether to return headings with the results (`SQLColumnInformation` option only)

`SQLColumns`, `SQLColumnNames`, and `SQLColumnInformation` options.

The option `"ShowColumnHeadings"` can be used with `SQLColumnInformation` to return the column headings.

```
In[97]:= SQLColumnInformation[conn, "SA%", "ShowColumnHeadings" → True] // TableForm
```

Out[97]=

TABLE_CAT	TABLE_SCHEM	TABLE_NAME	COLUMN_NAME	DATA_TYPE	TYPE_NAME	COLUMN_SIZE	BUFFER_LENGTH	DECIMAL_DIGITS
Null	PUBLIC	SAMPLETABLE1	ENTRY	4	INTEGER	Null	4	Null
Null	PUBLIC	SAMPLETABLE1	VALUE	8	DOUBLE	Null	8	Null
Null	PUBLIC	SAMPLETABLE1	NAME	12	VARCHAR	2 147 483 647	Null	Null

This closes the connection.

```
In[98]:= CloseSQLConnection[conn]
```

If the database was designed with particular schema and catalogs, you can also select columns by using the `"Catalog"` and `"Schema"` options.

Column Representation

`SQLColumn` expressions hold information about the columns in a database.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

`SQLColumn[{`*table*`,`*col*`},`*opts*`]`	expression that represents a column in an SQL table

Object for representing a column.

`SQLColumn` accepts a number of options.

option name	*default value*	
`"DataTypeName"`	`None`	type of the entry
`"Nullable"`	`None`	whether the entry can be null
`"DataLength"`	`None`	maximum length for variable length data

`SQLColumn` options.

Here is an example demonstrating `SQLColumn` expressions. This loads *DatabaseLink* and connects to the *demo* database.

```
In[99]:= Needs["DatabaseLink`"];
         conn = OpenSQLConnection["demo"];
```

`SQLColumns` returns a list of the columns in the database as `SQLColumn` expressions. In this example a pattern is given to pick out just the *SAMPLETABLE1* table.

```
In[101]:= SQLColumns[conn, "SAMPLETABLE1"]

Out[101]= {SQLColumn[{SAMPLETABLE1, ENTRY}, DataTypeName → INTEGER, Nullable → 1, DataLength → Null],
           SQLColumn[{SAMPLETABLE1, VALUE}, DataTypeName → DOUBLE, Nullable → 1, DataLength → Null],
           SQLColumn[{SAMPLETABLE1, NAME}, DataTypeName → VARCHAR, Nullable → 1, DataLength → 2 147 483 647]}
```

This closes the connection.

```
In[102]:= CloseSQLConnection[conn]
```

`SQLColumn` expressions can also be used in commands as discussed in "Selecting Data" and "Creating Tables". "Creating Tables" discusses one particularly important use.

Data Types

This tutorial discusses how to retrieve information about data types. When you create a table, you will need to refer to these data types.

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

`SQLDataTypeNames[`*conn*`]`	list all data type names within a data source
`SQLDataTypeInformation[`*conn*`]`	list all data type information within a data source

Functions for retrieving information about data types.

This loads *DatabaseLink* and connects to the *demo* database.

```
In[103]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

`SQLDataTypeNames` returns a list of the data type names within a database.

```
In[105]:= SQLDataTypeNames[conn]

Out[105]= {TINYINT, BIGINT, LONGVARBINARY, VARBINARY, BINARY, LONGVARCHAR,
           CHAR, NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE,
           VARCHAR, VARCHAR_IGNORECASE, BOOLEAN, DATE, TIME, TIMESTAMP, OTHER}
```

`SQLDataTypeInformation` returns more complete information about the data types.

```
In[106]:= SQLDataTypeInformation[conn] // TableForm
```

Out[106]=

TINYINT	-6	3	Null	Null	Null	1	False	3	False	False	False	TINYINT
BIGINT	-5	19	Null	Null	Null	1	False	3	False	False	True	BIGINT
LONGVARBINARY	-4	2 147 483 647	'	'	Null	1	False	3	Null	Null	Null	LONGVARBINARY
VARBINARY	-3	2 147 483 647	'	'	Null	1	False	3	Null	Null	Null	VARBINARY
BINARY	-2	2 147 483 647	'	'	Null	1	False	3	Null	Null	Null	BINARY
LONGVARCHAR	-1	2 147 483 647	'	'	Null	1	True	3	Null	Null	Null	LONGVARCHAR
CHAR	1	2 147 483 647	'	'	LENGTH	1	True	3	Null	Null	Null	CHAR
NUMERIC	2	646 456 993	Null	Null	PRECISION,SCALE	1	False	3	False	False	False	NUMERIC
DECIMAL	3	646 456 993	Null	Null	PRECISION,SCALE	1	False	3	False	False	False	DECIMAL
INTEGER	4	10	Null	Null	Null	1	False	3	False	False	True	INTEGER
SMALLINT	5	5	Null	Null	Null	1	False	3	False	False	False	SMALLINT
FLOAT	6	17	Null	Null	PRECISION	1	False	3	False	False	False	FLOAT
REAL	7	17	Null	Null	Null	1	False	3	False	False	False	REAL
DOUBLE	8	17	Null	Null	Null	1	False	3	False	False	False	DOUBLE
VARCHAR	12	2 147 483 647	'	'	LENGTH	1	False	3	Null	Null	Null	VARCHAR
VARCHAR_IGNORECASE	12	2 147 483 647	'	'	LENGTH	1	False	3	Null	Null	Null	VARCHAR_IGNORECASE
BOOLEAN	16	1	Null	Null	Null	1	False	3	Null	Null	Null	BOOLEAN
DATE	91	10	'	'	Null	1	False	3	Null	Null	Null	DATE

TIME	92	8	'	'	Null	1	False	3	Null	Null	Null	TIME
TIMESTAMP	93	29	'	'	PRECISION	1	False	3	Null	Null	Null	TIMESTAMP
OTHER	1111	2 147 483 647	'	'	Null	1	True	0	Null	Null	Null	OTHER

`SQLDataTypeInformation` takes a single option: `"ShowColumnHeadings"`. This returns the column headings.

In[107]:=
```
SQLDataTypeInformation[conn, "ShowColumnHeadings" → True] // TableForm
```

Out[107]=

TYPE_NAME	DATA_TYPE	PRECISION	LITERAL_PREFIX	LITERAL_SUFFIX	CREATE_PARAMS	NULLABLE
TINYINT	-6	3	Null	Null	Null	1
BIGINT	-5	19	Null	Null	Null	1
LONGVARBINARY	-4	2 147 483 647	'	'	Null	1
VARBINARY	-3	2 147 483 647	'	'	Null	1
BINARY	-2	2 147 483 647	'	'	Null	1
LONGVARCHAR	-1	2 147 483 647	'	'	Null	1
CHAR	1	2 147 483 647	'	'	LENGTH	1
NUMERIC	2	646 456 993	Null	Null	PRECISION,SCALE	1
DECIMAL	3	646 456 993	Null	Null	PRECISION,SCALE	1
INTEGER	4	10	Null	Null	Null	1
SMALLINT	5	5	Null	Null	Null	1
FLOAT	6	17	Null	Null	PRECISION	1
REAL	7	17	Null	Null	Null	1
DOUBLE	8	17	Null	Null	Null	1
VARCHAR	12	2 147 483 647	'	'	LENGTH	1
VARCHAR_IGNORECASE	12	2 147 483 647	'	'	LENGTH	1
BOOLEAN	16	1	Null	Null	Null	1
DATE	91	10	'	'	Null	1
TIME	92	8	'	'	Null	1
TIMESTAMP	93	29	'	'	PRECISION	1
OTHER	1111	2 147 483 647	'	'	Null	1

This closes the connection.

In[108]:=
```
CloseSQLConnection[conn]
```

More information on working with data types is provided in "Data Type Mapping".

Schema and Catalogs

Database schema and catalogs can be used to hold collections of database components and objects suitable for particular users. They can be particularly useful when working with large databases. The functions `SQLSchemaNames` and `SQLCatalogNames` can be used to learn the names of the schema and catalogs in the database. These can be used with the `"Schema"` and `"Catalog"` options to `SQLTableNames`, `SQLTableInformation`, `SQLTables`, `SQLColumnNames`, `SQLColumnInformation`, and `SQLColumns` to focus attention on particular parts of the database.

SQLCatalogNames[*conn*]	list all the catalogs used in a data source
SQLSchemaNames[*conn*]	list all the schema used in a data source
SQLSchemaInformation[*conn*]	returns information about the schema used in a data source

Listing catalogs and schema.

If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the DatabaseExamples` package, as described in "Using the Example Databases".

First, the *DatabaseLink* package is loaded and a connection is made to the *publisher* example database.

```
In[109]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection["publisher"];
```

This returns the schema names for the connection.

```
In[111]:= SQLSchemaNames[conn]
Out[111]= {INFORMATION_SCHEMA, PUBLIC}
```

SQLSchemaInformation returns more information about the database schema.

```
In[112]:= SQLSchemaInformation[conn]
Out[112]= {{INFORMATION_SCHEMA, Null, False}, {PUBLIC, Null, True}}
```

This returns the catalog names; for this database there are not catalogs.

```
In[113]:= SQLCatalogNames[conn]
Out[113]= {}
```

```
In[114]:= CloseSQLConnection[conn]
```

Data Commands

Comparing *Mathematica* and SQL Queries

DatabaseLink provides two styles of commands for working with data: one for those who are familiar with *Mathematica* and the other for those who are familiar with SQL. *Mathematica* style requires less knowledge of SQL. However, the *Mathematica* commands do not give complete coverage; thus, for more advanced queries, SQL-style commands may be preferred. The latter may also be desirable if you already have a knowledge of SQL.

Mathematica-Style Queries

DatabaseLink offers a number of functions for *Mathematica*-style queries.

- `SQLSelect`
- `SQLUpdate`
- `SQLInsert`
- `SQLDelete`
- `SQLCreateTable`
- `SQLDropTable`
- `SQLMemberQ`
- `SQLStringMatchQ`

The first six functions interact with the database. `SQLMemberQ` and `SQLStringMatchQ` are used for testing data in queries with conditions.

SQL-Style Queries

DatabaseLink can work with databases with raw SQL statements. This is useful if you already have a knowledge of SQL. Statements can be used to select data, create tables, insert data, update data, remove data, and drop tables. Typically these statements are passed to a command, `SQLExecute`. The statement used by `SQLExecute` is a string that can contain all argu-

`SQLExecute`

ments. However, it is also possible to give the arguments separately, which makes the statement a prepared statement. `SQLExecute` can also be used to execute a batch of prepared statements with different arguments, as described in Performance: Batch Operation.

Selecting Data

`SQLSelect` selects and returns data from a database. An alternative, using raw SQL, is described in "Selecting Data with Raw SQL".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

`SQLSelect[`*conn*`,`*table*`,`*opts*`]`	select all data from the table
`SQLSelect[`*conn*`,{`*tables*`},{`*columns*`}]`	select data in certain columns from the table
`SQLSelect[`*conn*`,{`*tables*`},{`*columns*`},`*condition*`,`*opts*`]`	select data in certain columns from the table meeting the condition

Retrieving data from a database.

This loads *DatabaseLink* and connects to the *publisher* database.

```
In[115]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "publisher"];
```

This retrieves all data within the table *ROYSCHED*.

```
In[117]:= SQLSelect[conn, "ROYSCHED"]

Out[117]= {{BS1011, 0, 5000, 0.1}, {BS1011, 5001, 50000, 0.12}, {CP5018, 0, 2000, 0.1},
  {CP5018, 2001, 4000, 0.12}, {CP5018, 4001, 50000, 0.16}, {BS1001, 0, 1000, 0.1},
  {BS1001, 1001, 5000, 0.12}, {BS1001, 5001, 7000, 0.16}, {BS1001, 7001, 50000, 0.18},
  {PS9999, 0, 50000, 0.1}, {PY2002, 0, 1000, 0.1}, {PY2002, 1001, 5000, 0.12},
  {PY2002, 5001, 50000, 0.14}, {PY2003, 0, 2000, 0.1}, {PY2003, 2001, 5000, 0.12},
  {PY2003, 5001, 50000, 0.14}, {UK3004, 0, 1000, 0.1}, {UK3004, 1001, 2000, 0.12},
  {UK3004, 2001, 6000, 0.14}, {UK3004, 6001, 8000, 0.18}, {UK3004, 8001, 50000, 0.2},
  {CK4005, 0, 2000, 0.1}, {CK4005, 2001, 6000, 0.12}, {CK4005, 6001, 8000, 0.16},
  {CK4005, 8001, 50000, 0.16}, {CP5010, 0, 5000, 0.1}, {CP5010, 5001, 50000, 0.12},
  {PY2012, 0, 5000, 0.1}, {PY2012, 5001, 50000, 0.12}, {PY2013, 0, 5000, 0.1},
  {PY2013, 5001, 50000, 0.12}, {UK3006, 0, 1000, 0.1}, {UK3006, 1001, 2000, 0.12},
  {UK3006, 2001, 6000, 0.14}, {UK3006, 6001, 8000, 0.18}, {UK3006, 8001, 50000, 0.2},
  {BS1014, 0, 4000, 0.1}, {BS1014, 4001, 8000, 0.12}, {BS1014, 8001, 50000, 0.14},
  {UK3015, 0, 2000, 0.1}, {UK3015, 2001, 4000, 0.12}, {UK3015, 4001, 8000, 0.14},
  {UK3015, 8001, 12000, 0.16}, {CK4016, 0, 5000, 0.1}, {CK4016, 5001, 15000, 0.12},
  {CK4017, 0, 2000, 0.1}, {CK4017, 2001, 8000, 0.12}, {CK4017, 8001, 16000, 0.14},
  {BS1007, 0, 5000, 0.1}, {BS1007, 5001, 50000, 0.12}, {PY2008, 0, 50000, 0.1}}
```

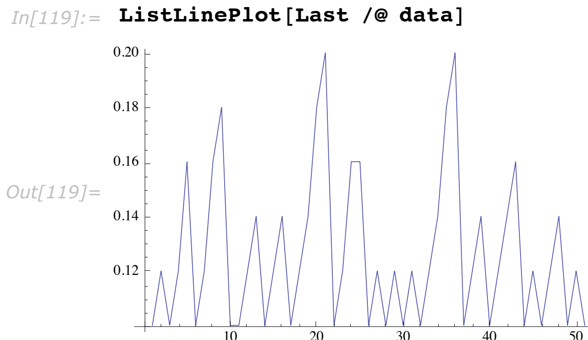
The third parameter of `SQLSelect` can be used to select only certain columns. In this example, only the *TITLE_ID* and *ROYALTY* columns are selected.

```
In[118]:= data = SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"}]

Out[118]= {{BS1011, 0.1}, {BS1011, 0.12}, {CP5018, 0.1}, {CP5018, 0.12}, {CP5018, 0.16}, {BS1001, 0.1},
  {BS1001, 0.12}, {BS1001, 0.16}, {BS1001, 0.18}, {PS9999, 0.1}, {PY2002, 0.1},
  {PY2002, 0.12}, {PY2002, 0.14}, {PY2003, 0.1}, {PY2003, 0.12}, {PY2003, 0.14},
  {UK3004, 0.1}, {UK3004, 0.12}, {UK3004, 0.14}, {UK3004, 0.18}, {UK3004, 0.2}, {CK4005, 0.1},
  {CK4005, 0.12}, {CK4005, 0.16}, {CK4005, 0.16}, {CP5010, 0.1}, {CP5010, 0.12}, {PY2012, 0.1},
  {PY2012, 0.12}, {PY2013, 0.1}, {PY2013, 0.12}, {UK3006, 0.1}, {UK3006, 0.12}, {UK3006, 0.14},
  {UK3006, 0.18}, {UK3006, 0.2}, {BS1014, 0.1}, {BS1014, 0.12}, {BS1014, 0.14}, {UK3015, 0.1},
  {UK3015, 0.12}, {UK3015, 0.14}, {UK3015, 0.16}, {CK4016, 0.1}, {CK4016, 0.12},
  {CK4017, 0.1}, {CK4017, 0.12}, {CK4017, 0.14}, {BS1007, 0.1}, {BS1007, 0.12}, {PY2008, 0.1}}
```

The results of the database operation can immediately be used in *Mathematica*.

```
In[119]:= ListLinePlot[Last /@ data]
```

Out[119]=



There are a number of options that can be given to `SQLSelect`.

option name	*default value*	
`"SortingColumns"`	`None`	how to sort the data
`"Distinct"`	`False`	whether to return only distinct results
`"GetAsStrings"`	`False`	whether to return the results as strings
`"MaxRows"`	`Automatic`	set the maximum number of rows returned
`"ShowColumnHeadings"`	`False`	whether to return headings with the results
`"Timeout"`	`Automatic`	set the timeout for a query

Options of `SQLSelect`.

It is possible to select data from multiple columns in multiple tables. You can select multiple tables by giving a second argument that is a list of the table names. A list of column names should be used as the third parameter as shown previously. You can also associate a specific table with a column by pairing a column name with a table name in a list in the third argument.

`"MaxRows"`

This is important if the same column name is used in more than one table. The following example of a data join generates an outer product of the data in the two tables and it uses the option "MaxRows" to show only the first five results.

```
In[120]:= SQLSelect[conn, {"TITLES", "ROYSCHED"}, {{"TITLES", "TITLE"},
            {"TITLES", "TITLE_ID"}, {"ROYSCHED", "TITLE_ID"}, {"ROYSCHED", "ROYALTY"}},
           "MaxRows" → 5, "ShowColumnHeadings" → True] // TableForm

Out[120]= TITLE                      TITLE_ID TITLE_ID ROYALTY
          Designer Class Action Suits BS1001   BS1011   0.1
          Designer Class Action Suits BS1001   BS1011   0.12
          Designer Class Action Suits BS1001   CP5018   0.1
          Designer Class Action Suits BS1001   CP5018   0.12
          Designer Class Action Suits BS1001   CP5018   0.16
```

The following example repeats the previous query, adding a condition that the *TITLE_ID* in the two tables must be equal. Using a condition is often a useful way to narrow the search results.

```
In[121]:= SQLSelect[conn, {"TITLES", "ROYSCHED"}, {{"TITLES", "TITLE"},
            {"TITLES", "TITLE_ID"}, {"ROYSCHED", "TITLE_ID"}, {"ROYSCHED", "ROYALTY"}},
           SQLColumn[{"TITLES", "TITLE_ID"}] == SQLColumn[{"ROYSCHED", "TITLE_ID"}],
           "MaxRows" → 5, "ShowColumnHeadings" → True] // TableForm

Out[121]= TITLE                            TITLE_ID TITLE_ID ROYALTY
          Designer Class Action Suits       BS1001   BS1001   0.1
          Designer Class Action Suits       BS1001   BS1001   0.12
          Designer Class Action Suits       BS1001   BS1001   0.16
          Designer Class Action Suits       BS1001   BS1001   0.18
          Self Hypnosis: A Beginner's Guide PY2002   PY2002   0.1
```

You may specify that a column value must be between certain values.

```
In[122]:= SQLSelect[conn, "ROYSCHED",
           {"TITLE_ID", "ROYALTY"}, .10 < SQLColumn["ROYALTY"] < .15]

Out[122]= {{BS1011, 0.12}, {CP5018, 0.12}, {BS1001, 0.12}, {PY2002, 0.12}, {PY2002, 0.14}, {PY2003, 0.12},
           {PY2003, 0.14}, {UK3004, 0.12}, {UK3004, 0.14}, {CK4005, 0.12}, {CP5010, 0.12}, {PY2012, 0.12},
           {PY2013, 0.12}, {UK3006, 0.12}, {UK3006, 0.14}, {BS1014, 0.12}, {BS1014, 0.14}, {UK3015, 0.12},
           {UK3015, 0.14}, {CK4016, 0.12}, {CK4017, 0.12}, {CK4017, 0.14}, {BS1007, 0.12}}
```

```
In[123]:= SQLSelect[conn, "ROYSCHED",
           {"TITLE_ID", "ROYALTY"}, .13 > SQLColumn["ROYALTY"] > .10]

Out[123]= {{BS1011, 0.12}, {CP5018, 0.12}, {BS1001, 0.12}, {PY2002, 0.12}, {PY2003, 0.12}, {UK3004, 0.12},
           {CK4005, 0.12}, {CP5010, 0.12}, {PY2012, 0.12}, {PY2013, 0.12}, {UK3006, 0.12},
           {BS1014, 0.12}, {UK3015, 0.12}, {CK4016, 0.12}, {CK4017, 0.12}, {BS1007, 0.12}}
```

You may specify that a column value must be equal to a certain value.

```
In[124]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"}, SQLColumn["ROYALTY"] == .12]

Out[124]= {{BS1011, 0.12}, {CP5018, 0.12}, {BS1001, 0.12}, {PY2002, 0.12}, {PY2003, 0.12}, {UK3004, 0.12},
           {CK4005, 0.12}, {CP5010, 0.12}, {PY2012, 0.12}, {PY2013, 0.12}, {UK3006, 0.12},
           {BS1014, 0.12}, {UK3015, 0.12}, {CK4016, 0.12}, {CK4017, 0.12}, {BS1007, 0.12}}
```

You may specify that a column value must not be equal to a certain value.

```
In[125]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"}, SQLColumn["ROYALTY"] != .12]
Out[125]= {{BS1011, 0.1}, {CP5018, 0.1}, {CP5018, 0.16}, {BS1001, 0.1}, {BS1001, 0.16}, {BS1001, 0.18},
  {PS9999, 0.1}, {PY2002, 0.1}, {PY2002, 0.14}, {PY2003, 0.1}, {PY2003, 0.14}, {UK3004, 0.1},
  {UK3004, 0.14}, {UK3004, 0.18}, {UK3004, 0.2}, {CK4005, 0.1}, {CK4005, 0.16},
  {CK4005, 0.16}, {CP5010, 0.1}, {PY2012, 0.1}, {PY2013, 0.1}, {UK3006, 0.1}, {UK3006, 0.14},
  {UK3006, 0.18}, {UK3006, 0.2}, {BS1014, 0.1}, {BS1014, 0.14}, {UK3015, 0.1}, {UK3015, 0.14},
  {UK3015, 0.16}, {CK4016, 0.1}, {CK4017, 0.1}, {CK4017, 0.14}, {BS1007, 0.1}, {PY2008, 0.1}}
```

You may specify that a column value must be greater than a certain value.

```
In[126]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"}, SQLColumn["ROYALTY"] > .12]
Out[126]= {{CP5018, 0.16}, {BS1001, 0.16}, {BS1001, 0.18}, {PY2002, 0.14}, {PY2003, 0.14}, {UK3004, 0.14},
  {UK3004, 0.18}, {UK3004, 0.2}, {CK4005, 0.16}, {CK4005, 0.16}, {UK3006, 0.14}, {UK3006, 0.18},
  {UK3006, 0.2}, {BS1014, 0.14}, {UK3015, 0.14}, {UK3015, 0.16}, {CK4017, 0.14}}
```

You may specify that a column value must be less than a certain value.

```
In[127]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"}, SQLColumn["ROYALTY"] < .12]
Out[127]= {{BS1011, 0.1}, {CP5018, 0.1}, {BS1001, 0.1}, {PS9999, 0.1}, {PY2002, 0.1}, {PY2003, 0.1},
  {UK3004, 0.1}, {CK4005, 0.1}, {CP5010, 0.1}, {PY2012, 0.1}, {PY2013, 0.1}, {UK3006, 0.1},
  {BS1014, 0.1}, {UK3015, 0.1}, {CK4016, 0.1}, {CK4017, 0.1}, {BS1007, 0.1}, {PY2008, 0.1}}
```

You may specify that a column value must be greater than or equal to a certain value.

```
In[128]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"}, SQLColumn["ROYALTY"] >= .12]
Out[128]= {{BS1011, 0.12}, {CP5018, 0.12}, {CP5018, 0.16}, {BS1001, 0.12}, {BS1001, 0.16}, {BS1001, 0.18},
  {PY2002, 0.12}, {PY2002, 0.14}, {PY2003, 0.12}, {PY2003, 0.14}, {UK3004, 0.12},
  {UK3004, 0.14}, {UK3004, 0.18}, {UK3004, 0.2}, {CK4005, 0.12}, {CK4005, 0.16}, {CK4005, 0.16},
  {CP5010, 0.12}, {PY2012, 0.12}, {PY2013, 0.12}, {UK3006, 0.12}, {UK3006, 0.14},
  {UK3006, 0.18}, {UK3006, 0.2}, {BS1014, 0.12}, {BS1014, 0.14}, {UK3015, 0.12}, {UK3015, 0.14},
  {UK3015, 0.16}, {CK4016, 0.12}, {CK4017, 0.12}, {CK4017, 0.14}, {BS1007, 0.12}}
```

You may specify that a column value must match a certain pattern using the metacharacters '%' for matching zero or more characters and '_' for matching a single character.

```
In[129]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"},
  SQLStringMatchQ[SQLColumn["TITLE_ID"], "C%"]]
Out[129]= {{CP5018, 0.1}, {CP5018, 0.12}, {CP5018, 0.16}, {CK4005, 0.1},
  {CK4005, 0.12}, {CK4005, 0.16}, {CK4005, 0.16}, {CP5010, 0.1}, {CP5010, 0.12},
  {CK4016, 0.1}, {CK4016, 0.12}, {CK4017, 0.1}, {CK4017, 0.12}, {CK4017, 0.14}}
```

```
In[130]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"},
  SQLStringMatchQ[SQLColumn["TITLE_ID"], "_S%"]]
Out[130]= {{BS1011, 0.1}, {BS1011, 0.12}, {BS1001, 0.1}, {BS1001, 0.12}, {BS1001, 0.16}, {BS1001, 0.18},
  {PS9999, 0.1}, {BS1014, 0.1}, {BS1014, 0.12}, {BS1014, 0.14}, {BS1007, 0.1}, {BS1007, 0.12}}
```

You may specify that a column value must be contained as a member of a list.

```
In[131]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"},
  SQLMemberQ[{.14, .16}, SQLColumn["ROYALTY"]]]
Out[131]= {{CP5018, 0.16}, {BS1001, 0.16}, {PY2002, 0.14}, {PY2003, 0.14}, {UK3004, 0.14}, {CK4005, 0.16},
  {CK4005, 0.16}, {UK3006, 0.14}, {BS1014, 0.14}, {UK3015, 0.14}, {UK3015, 0.16}, {CK4017, 0.14}}
```

You may specify that a column value must be less than or equal to a certain value.

```
In[132]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"}, SQLColumn["ROYALTY"] <= .12]

Out[132]= {{BS1011, 0.1}, {BS1011, 0.12}, {CP5018, 0.1}, {CP5018, 0.12}, {BS1001, 0.1}, {BS1001, 0.12},
  {PS9999, 0.1}, {PY2002, 0.1}, {PY2002, 0.12}, {PY2003, 0.1}, {PY2003, 0.12}, {UK3004, 0.1},
  {UK3004, 0.12}, {CK4005, 0.1}, {CK4005, 0.12}, {CP5010, 0.1}, {CP5010, 0.12}, {PY2012, 0.1},
  {PY2012, 0.12}, {PY2013, 0.1}, {PY2013, 0.12}, {UK3006, 0.1}, {UK3006, 0.12}, {BS1014, 0.1},
  {BS1014, 0.12}, {UK3015, 0.1}, {UK3015, 0.12}, {CK4016, 0.1}, {CK4016, 0.12},
  {CK4017, 0.1}, {CK4017, 0.12}, {BS1007, 0.1}, {BS1007, 0.12}, {PY2008, 0.1}}
```

You may also combine any conditions using `And` or `Or`.

```
In[133]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "LORANGE", "ROYALTY"},
  SQLColumn["ROYALTY"] == .12 && SQLColumn["LORANGE"] > 1000]

Out[133]= {{BS1011, 5001, 0.12}, {CP5018, 2001, 0.12}, {BS1001, 1001, 0.12}, {PY2002, 1001, 0.12},
  {PY2003, 2001, 0.12}, {UK3004, 1001, 0.12}, {CK4005, 2001, 0.12}, {CP5010, 5001, 0.12},
  {PY2012, 5001, 0.12}, {PY2013, 5001, 0.12}, {UK3006, 1001, 0.12}, {BS1014, 4001, 0.12},
  {UK3015, 2001, 0.12}, {CK4016, 5001, 0.12}, {CK4017, 2001, 0.12}, {BS1007, 5001, 0.12}}
```

```
In[134]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"},
  SQLColumn["ROYALTY"] == .12 || SQLColumn["ROYALTY"] == .14]

Out[134]= {{BS1011, 0.12}, {CP5018, 0.12}, {BS1001, 0.12}, {PY2002, 0.12}, {PY2002, 0.14}, {PY2003, 0.12},
  {PY2003, 0.14}, {UK3004, 0.12}, {UK3004, 0.14}, {CK4005, 0.12}, {CP5010, 0.12}, {PY2012, 0.12},
  {PY2013, 0.12}, {UK3006, 0.12}, {UK3006, 0.14}, {BS1014, 0.12}, {BS1014, 0.14}, {UK3015, 0.12},
  {UK3015, 0.14}, {CK4016, 0.12}, {CK4017, 0.12}, {CK4017, 0.14}, {BS1007, 0.12}}
```

The option `"GetAsStrings"` can retrieve data without converting it to a *Mathematica* type. This repeats the previous query without converting the data.

```
In[135]:= SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"},
   SQLColumn["ROYALTY"] == .12 || SQLColumn["ROYALTY"] == .14,
   "GetAsStrings" → True] // InputForm

Out[135]= {{"BS1011", "0.12"}, {"CP5018", "0.12"},
  {"BS1001", "0.12"}, {"PY2002", "0.12"},
  {"PY2002", "0.14"}, {"PY2003", "0.12"},
  {"PY2003", "0.14"}, {"UK3004", "0.12"},
  {"UK3004", "0.14"}, {"CK4005", "0.12"},
  {"CP5010", "0.12"}, {"PY2012", "0.12"},
  {"PY2013", "0.12"}, {"UK3006", "0.12"},
  {"UK3006", "0.14"}, {"BS1014", "0.12"},
  {"BS1014", "0.14"}, {"UK3015", "0.12"},
  {"UK3015", "0.14"}, {"CK4016", "0.12"},
  {"CK4017", "0.12"}, {"CK4017", "0.14"},
  {"BS1007", "0.12"}}
```

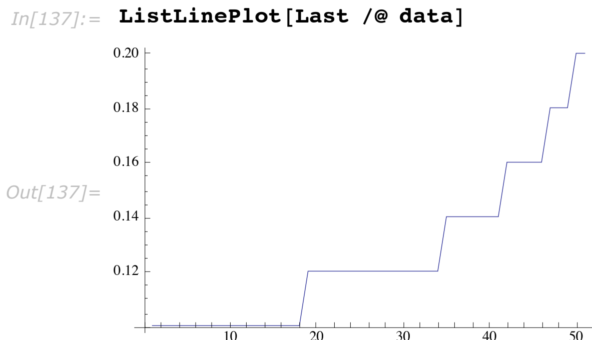
You may also use the option `"SortingColumns"` to specify how to sort the data. This option takes a list of rules. The left side of the rule specifies the column and the right side specifies whether to sort that data in ascending or descending order. The first item in the list takes precedence over the supplemental items.

```
In[136]:= data = SQLSelect[conn, "ROYSCHED", {"TITLE_ID", "ROYALTY"}, "SortingColumns" →
            {SQLColumn["ROYALTY"] -> "Ascending", SQLColumn["TITLE_ID"] -> "Ascending"}]

Out[136]= {{BS1001, 0.1}, {BS1007, 0.1}, {BS1011, 0.1}, {BS1014, 0.1}, {CK4005, 0.1}, {CK4016, 0.1},
           {CK4017, 0.1}, {CP5010, 0.1}, {CP5018, 0.1}, {PS9999, 0.1}, {PY2002, 0.1}, {PY2003, 0.1},
           {PY2008, 0.1}, {PY2012, 0.1}, {PY2013, 0.1}, {UK3004, 0.1}, {UK3006, 0.1}, {UK3015, 0.1},
           {BS1001, 0.12}, {BS1007, 0.12}, {BS1011, 0.12}, {BS1014, 0.12}, {CK4005, 0.12}, {CK4016, 0.12},
           {CK4017, 0.12}, {CP5010, 0.12}, {CP5018, 0.12}, {PY2002, 0.12}, {PY2003, 0.12}, {PY2012, 0.12},
           {PY2013, 0.12}, {UK3004, 0.12}, {UK3006, 0.12}, {UK3015, 0.12}, {BS1014, 0.14},
           {CK4017, 0.14}, {PY2002, 0.14}, {PY2003, 0.14}, {UK3004, 0.14}, {UK3006, 0.14},
           {UK3015, 0.14}, {BS1001, 0.16}, {CK4005, 0.16}, {CK4005, 0.16}, {CP5018, 0.16},
           {UK3015, 0.16}, {BS1001, 0.18}, {UK3004, 0.18}, {UK3006, 0.18}, {UK3004, 0.2}, {UK3006, 0.2}}
```

The following plot shows that the data is now sorted.

```
In[137]:= ListLinePlot[Last /@ data]
```

Out[137]=



The option "Timeout" can be used to cancel a query if it takes too long to execute.

This closes the connection.

```
In[138]:= CloseSQLConnection[conn]
```

The details of how *Mathematica* expressions are mapped to types stored in the database is discussed in "Data Type Mapping".

Creating Tables

`SQLCreateTable` creates a new table in a database. An alternative, using raw SQL, is described in "Creating Tables with Raw SQL".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

When creating a table, the result of `SQLCreateTable` is an integer specifying the number of rows affected by the query. If the table is created correctly, this integer will always be zero as no rows are affected when creating a new table.

SQLCreateTable[*conn*,*table*,{*columns*},*opts*]	create an SQL table

Creating a table in a database.

Here is an example that creates a table.

This loads *DatabaseLink* and connects to the *demo* database.

```
In[139]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

SQLCreateTable creates a table. The columns are given as a list of SQLColumn expressions. In the following example, a new table, *DATATYPESTABLE*, is created that has one column for each of the data types returned from SQLDataTypeNames. The column, *TINYINTCOL*, is configured so that it cannot be set to Null. However, each binary column can be set to Null. The database default for "Nullable" is used for every other column that does not specify the "Nullable" option. The character-based columns are limited to a specific data length; other columns use the default data length for their type.

```
In[141]:= SQLDataTypeNames[conn]

Out[141]= {TINYINT, BIGINT, LONGVARBINARY, VARBINARY, BINARY, LONGVARCHAR,
           CHAR, NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE,
           VARCHAR, VARCHAR_IGNORECASE, BOOLEAN, DATE, TIME, TIMESTAMP, OTHER}
```

```
In[142]:= SQLCreateTable[conn, "DATATYPESTABLE",
           {
            SQLColumn["TINYINTCOL", "DataTypeName" -> "TINYINT", "Nullable" → False],
            SQLColumn["SMALLINTCOL", "DataTypeName" -> "SMALLINT"],
            SQLColumn["INTEGERCOL", "DataTypeName" -> "INTEGER"],
            SQLColumn["BIGINTCOL", "DataTypeName" -> "BIGINT"],
            SQLColumn["NUMERICCOL", "DataTypeName" -> "NUMERIC"],
            SQLColumn["DECIMALCOL", "DataTypeName" -> "DECIMAL"],
            SQLColumn["FLOATCOL", "DataTypeName" -> "FLOAT"],
            SQLColumn["REALCOL", "DataTypeName" -> "REAL"],
            SQLColumn["DOUBLECOL", "DataTypeName" -> "DOUBLE"],
            SQLColumn["BITCOL", "DataTypeName" -> "BIT"],
            SQLColumn["LONGVARBINARYCOL",
             "DataTypeName" -> "LONGVARBINARY", "Nullable" → True],
            SQLColumn["VARBINARYCOL", "DataTypeName" -> "VARBINARY", "Nullable" → True],
            SQLColumn["BINARYCOL", "DataTypeName" -> "BINARY", "Nullable" → True],
            SQLColumn["LONGVARCHARCOL", "DataTypeName" -> "LONGVARCHAR"],
            SQLColumn["VARCHARCOL",
             "DataTypeName" -> "VARCHAR", "Nullable" → True, "DataLength" → 5],
            SQLColumn["CHARCOL", "DataTypeName" -> "CHAR",
             "Nullable" → True, "DataLength" → 3],
            SQLColumn["DATECOL", "DataTypeName" -> "DATE"],
            SQLColumn["TIMECOL", "DataTypeName" -> "TIME"],
            SQLColumn["TIMESTAMPCOL", "DataTypeName" -> "TIMESTAMP"],
            SQLColumn["OBJECTCOL", "DataTypeName" -> "OBJECT", "Nullable" → True]
           }]

Out[142]= 0
```

`SQLTableNames` verifies that the table exists in the database.

```
In[143]:= SQLTableNames[conn, "DATATYPESTABLE"]
Out[143]= {DATATYPESTABLE}
```

`SQLColumnNames` verifies the columns in the table.

```
In[144]:= SQLColumnNames[conn, "DATATYPESTABLE"]
Out[144]= {{DATATYPESTABLE, TINYINTCOL}, {DATATYPESTABLE, SMALLINTCOL},
  {DATATYPESTABLE, INTEGERCOL}, {DATATYPESTABLE, BIGINTCOL}, {DATATYPESTABLE, NUMERICCOL},
  {DATATYPESTABLE, DECIMALCOL}, {DATATYPESTABLE, FLOATCOL}, {DATATYPESTABLE, REALCOL},
  {DATATYPESTABLE, DOUBLECOL}, {DATATYPESTABLE, BITCOL}, {DATATYPESTABLE, LONGVARBINARYCOL},
  {DATATYPESTABLE, VARBINARYCOL}, {DATATYPESTABLE, BINARYCOL}, {DATATYPESTABLE, LONGVARCHARCOL},
  {DATATYPESTABLE, VARCHARCOL}, {DATATYPESTABLE, CHARCOL}, {DATATYPESTABLE, DATECOL},
  {DATATYPESTABLE, TIMECOL}, {DATATYPESTABLE, TIMESTAMPCOL}, {DATATYPESTABLE, OBJECTCOL}}
```

`SQLCreateTable` accepts one option.

option name	*default value*	
`"Timeout"`	`Automatic`	set the timeout for a query

Option of `SQLCreateTable`.

`"Timeout"` can be used to cancel a query if it takes too long to execute.

This drops the table and closes the connection.

```
In[145]:= SQLDropTable[conn, "DATATYPESTABLE"];
          CloseSQLConnection[conn]
```

Certain databases support further options for columns, such as whether a column is a key or whether it auto-increments. If these options are desired, then a raw SQL statement should be used to create the table. "Creating Tables with Raw SQL" has some ideas and examples.

Inserting Data

`SQLInsert` inserts data into a database. An alternative, using raw SQL, is described in "Inserting Data with Raw SQL".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

The result of `SQLInsert` is an integer specifying the number of rows affected by the query. For a single insert this will be one, since you can only insert one row at a time. `SQLInsert` also supports a batch insert, as demonstrated in "Performance: Batch Operation".

SQLInsert[*conn*,*table*,{*columns*},{*values*},*opts*]	insert data into a database
SQLInsert[*conn*,*table*,{*columns*},{{*values*}},*opts*]	batch insert data into a database

Inserting data into a database.

Here is an example that inserts data. This loads *DatabaseLink* and connects to the *demo* database.

```
In[147]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

A new table, *TEST*, is created. The details of this command are described in "Creating Tables".

```
In[149]:= SQLCreateTable[conn, "TEST",
           {
            SQLColumn["COL1", "DataTypeName" -> "INTEGER"],
            SQLColumn["COL2", "DataTypeName" -> "DOUBLE"]
           }]
Out[149]= 0
```

SQLInsert inserts data into this table.

```
In[150]:= SQLInsert[conn, "TEST", {"COL1", "COL2"}, {10, 10.5}]
Out[150]= 1
```

SQLSelect verifies the data stored in the database.

```
In[151]:= SQLSelect[conn, "TEST"]
Out[151]= {{10, 10.5}}
```

Finally, a batch insert is carried out. The result is a list of the number of lines that are modified.

```
In[152]:= SQLInsert[conn, "TEST", {"COL1", "COL2"}, {{10, 10.5}, {20, 55.1}}]
Out[152]= {1, 1}
```

SQLSelect shows that there are now three rows in this table.

```
In[153]:= SQLSelect[conn, "TEST"]
Out[153]= {{10, 10.5}, {10, 10.5}, {20, 55.1}}
```

`SQLInsert` accepts one option.

option name	default value	
"Timeout"	Automatic	set the timeout for a query

Option of `SQLInsert`.

The option `"Timeout"` can be used to cancel a query if it takes too long to execute.

This drops the table and closes the connection.

```
In[154]:= SQLDropTable[conn, "TEST"];
          CloseSQLConnection[conn]
```

The details of how *Mathematica* expressions are mapped to types stored in the database is discussed in "Data Type Mapping".

Updating Data

`SQLUpdate` modifies data in a database. An alternative, using raw SQL, is described in "Updating Data with Raw SQL".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

The result of `SQLUpdate` is an integer specifying the number of rows affected by the query.

SQLUpdate[*conn*, *table*, {*columns*}, {*values*}, *opts*]	update data in a database
SQLUpdate[*conn*, *table*, {*columns*}, {*values*}, *condition*, *opts*]	update data in a database using a condition

Updating data in a database.

Here is an example that updates data. This loads *DatabaseLink* and connects to the *demo* database.

```
In[156]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

A new table, TEST, is created and data is inserted.

```
In[158]:= SQLCreateTable[conn, "TEST",
            {
             SQLColumn["COL1", "DataTypeName" -> "INTEGER"],
             SQLColumn["COL2", "DataTypeName" -> "DOUBLE"]
            }];
          SQLInsert[conn, "TEST", {"COL1", "COL2"}, {10, 10.5}];
```

SQLSelect shows the values in the table.

```
In[160]:= SQLSelect[conn, "TEST"]
Out[160]= {{10, 10.5}}
```

SQLUpdate updates the elements in the database and SQLSelect shows the result.

```
In[161]:= SQLUpdate[conn, "TEST", {"COL1", "COL2"}, {12, 12.5}];
          SQLSelect[conn, "TEST"]
Out[162]= {{12, 12.5}}
```

Typically, it is useful to set a condition for an update, with the condition specifying which rows should be updated. (For more information on conditions, see "Selecting Data".) In the following example, another row is inserted into the database.

```
In[163]:= SQLInsert[conn, "TEST", {"COL1", "COL2"}, {20, 20.5}];
          SQLSelect[conn, "TEST"]
Out[164]= {{12, 12.5}, {20, 20.5}}
```

Here an update is given for rows for which the entry in the first column is less than 15.

```
In[165]:= SQLUpdate[conn, "TEST", {"COL1", "COL2"}, {4, 1.1}, SQLColumn["COL1"] < 15 ];
          SQLSelect[conn, "TEST"]
Out[166]= {{20, 20.5}, {4, 1.1}}
```

SQLUpdate accepts one option.

option name	*default value*	
"Timeout"	Automatic	set the timeout for a query

Option of SQLUpdate.

The option "Timeout" can be used to cancel a query if it takes too long to execute.

This drops the table and closes the connection.

```
In[167]:= SQLDropTable[conn, "TEST"];
          CloseSQLConnection[conn]
```

Deleting Data

`SQLDelete` deletes data from a database. An alternative, using raw SQL, is described in "Deleting Data with Raw SQL".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

The result of `SQLDelete` is an integer specifying the number of rows affected by the query. Thus, if three rows are removed, the result is three, and if no rows are removed, the result is zero.

`SQLDelete[`*table*`]`	delete data from a database
`SQLDelete[`*table*`,`*condition*`]`	delete data from a database using a condition

Deleting data from a database.

Here is an example that deletes data. This loads *DatabaseLink* and connects to the *demo* database.

```
In[169]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

A new table, *TEST*, is created and data is inserted.

```
In[171]:= SQLCreateTable[conn, "TEST",
            {
             SQLColumn["COL1", "DataTypeName" -> "INTEGER"],
             SQLColumn["COL2", "DataTypeName" -> "DOUBLE"]
            }];
          SQLInsert[conn, "TEST", {"COL1", "COL2"}, {{10, 10.5}, {20, 17.5}}];
          SQLSelect[conn, "TEST"]
Out[173]= {{10, 10.5}, {20, 17.5}}
```

The following deletes all the data from the table. Two rows were deleted, and the result is two.

```
In[174]:= SQLDelete[conn, "TEST"]
Out[174]= 2
```

`SQLSelect` verifies that all the data has been removed from the table.

```
In[175]:= SQLSelect[conn, "TEST"]
Out[175]= {}
```

This restores the data in the database.

```
In[176]:= SQLInsert[conn, "TEST", {"COL1", "COL2"}, {{10, 10.5}, {20, 17.5}}];
          SQLSelect[conn, "TEST"]
Out[177]= {{10, 10.5}, {20, 17.5}}
```

Here, a condition is used in the `SQLDelete` command, so that only rows for which the entry in the first column is greater than 15 are deleted. This deletes one row, and hence the result is one.

```
In[178]:= stmt = SQLDelete[conn, "TEST", SQLColumn["COL1"] > 15]
Out[178]= 1
```

`SQLSelect` verifies that one row was removed from the table.

```
In[179]:= SQLSelect[conn, "TEST"]
Out[179]= {{10, 10.5}}
```

`SQLDelete` accepts one option.

option name	default value	
"Timeout"	Automatic	set the timeout for a query

Option of `SQLDelete`.

The option `"Timeout"` can be used to cancel a query if it takes too long to execute.

This drops the table and closes the connection.

```
In[180]:= SQLDropTable[conn, "TEST"];
          CloseSQLConnection[conn]
```

Dropping Tables

`SQLDropTable` drops tables from a database. An alternative, using raw SQL, is demonstrated in "Dropping Tables with Raw SQL".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

The result of `SQLDropTable` is an integer specifying the number of rows affected by the query.

`SQLDropTable[`*table*`]`	drop a table from a database

Dropping a table from a database.

Here is an example that drops a table. This loads *DatabaseLink* and connects to the *demo* database.

```
In[182]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

A new table, *TEST*, is created and data is inserted.

```
In[184]:= SQLCreateTable[conn, "TEST",
            {
             SQLColumn["COL1", "DataTypeName" -> "INTEGER"],
             SQLColumn["COL2", "DataTypeName" -> "DOUBLE"]
            }];
          SQLInsert[conn, "TEST", {"COL1", "COL2"}, {10, 10.5}];
```

This drops the table.

```
In[186]:= SQLDropTable[conn, "TEST"]
Out[186]= 0
```

`SQLTableNames` verifies that the table is removed from the database.

```
In[187]:= SQLTableNames[conn, "TEST"]
Out[187]= {}
```

`SQLDropTable` accepts one option.

option name	*default value*	
`"Timeout"`	`Automatic`	set the timeout for a query

Option of `SQLDropTable`.

The option `"Timeout"` can be used to cancel a query if it takes too long to execute.

This closes the connection.

```
In[188]:= CloseSQLConnection[conn]
```

SQLExecute

`SQLExecute` allows SQL statements to be executed. Statements can be used to select data, create tables, insert data, update data, remove data, and drop tables. The statement used by `SQLExecute` is a string that can contain all arguments. However, it is also possible to give the arguments separately, which makes the statement a prepared statement. `SQLExecute` can also be used to execute a batch of prepared statements with different arguments, as described in "Batch Input".

`SQLExecute[`*conn*`,`*statement*`,`*opts*…`]`	execute an SQL statement
`SQLExecute[`*conn*`,` *statement*`,{`*args*…`},`*opts*…`]`	execute a prepared statement with arguments
`SQLExecute[`*conn*`,` *statement*`,{{`*args*…`} …},`*opts*…`]`	execute a batch of prepared statement with different arguments

Executing SQL statements.

The following sections show how to use SQL statements to carry out different types of manipulations.

There are a number of options that can be given to `SQLExecute`.

option name	*default value*	
`"GetAsStrings"`	`False`	return the results as strings
`"MaxRows"`	`Automatic`	set the maximum number of rows returned
`"ShowColumnHeadings"`	`False`	whether to return headings with the results
`"Timeout"`	`Automatic`	set the timeout for a query

Options of `SQLExecute`.

Here is an example of these options. This loads *DatabaseLink* and connects to the *demo* database. If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `` DatabaseExamples` `` package, as described in "Using the Example Databases".

In[189]:=
```
Needs["DatabaseLink`"];
conn = OpenSQLConnection[ "demo"];
```

The option "GetAsStrings" can retrieve data without converting it to a *Mathematica* type.

```
In[191]:= SQLExecute[conn, "SELECT * FROM SAMPLETABLE1", "GetAsStrings" → True] // InputForm

Out[191]= {{"1", "5.6", "Day1"}, {"2", "5.9", "Day2"},
           {"3", "7.2", "Day3"}, {"4", "6.2", "Day4"},
           {"5", "6.0", "Day5"}}
```

The option "MaxRows" can limit the number of rows returned.

```
In[192]:= SQLExecute[conn, "SELECT * FROM SAMPLETABLE1", "MaxRows" → 2]

Out[192]= {{1, 5.6, Day1}, {2, 5.9, Day2}}
```

The option "ShowColumnHeadings" can retrieve the column headings with the results.

```
In[193]:= SQLExecute[conn, "SELECT * FROM SAMPLETABLE1",
            "ShowColumnHeadings" → True] // TableForm
```

Out[193]=

ENTRY	VALUE	NAME
1	5.6	Day1
2	5.9	Day2
3	7.2	Day3
4	6.2	Day4
5	6.	Day5

The option "Timeout" can be used to cancel a query if it takes too long to execute.

This closes the connection.

```
In[194]:= CloseSQLConnection[conn]
```

Argument Sequences in SQL-Style Queries

If you want to use one argument in an SQL statement that holds a sequence of several values, you can use `SQLArgument`. This is particularly useful for selects and inserts in tables that have many columns. With selects, you can dynamically specify multiple tables and columns, and with inserts you can dynamically specify multiple columns and values.

`SQLArgument`	a sequence of arguments to a command

Argument sequences.

To demonstrate this, load *DatabaseLink* and connect to the *publisher* database.

```
In[195]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "publisher"];
```

Now, you can execute a select query using `SQLArgument`. Notice how the statement refers to two arguments as `` `1` `` arguments. This makes the statement simpler since it saves having to number the arguments individually.

```
In[197]:= SQLExecute[conn, "SELECT `1` FROM `2`",
           { SQLArgument[SQLColumn["TITLE_ID"], SQLColumn["ROYALTY"]],
            SQLTable["ROYSCHED"]}]
Out[197]= {{BS1011, 0.1}, {BS1011, 0.12}, {CP5018, 0.1}, {CP5018, 0.12}, {CP5018, 0.16}, {BS1001, 0.1},
           {BS1001, 0.12}, {BS1001, 0.16}, {BS1001, 0.18}, {PS9999, 0.1}, {PY2002, 0.1},
           {PY2002, 0.12}, {PY2002, 0.14}, {PY2003, 0.1}, {PY2003, 0.12}, {PY2003, 0.14},
           {UK3004, 0.1}, {UK3004, 0.12}, {UK3004, 0.14}, {UK3004, 0.18}, {UK3004, 0.2}, {CK4005, 0.1},
           {CK4005, 0.12}, {CK4005, 0.16}, {CK4005, 0.16}, {CP5010, 0.1}, {CP5010, 0.12}, {PY2012, 0.1},
           {PY2012, 0.12}, {PY2013, 0.1}, {PY2013, 0.12}, {UK3006, 0.1}, {UK3006, 0.12}, {UK3006, 0.14},
           {UK3006, 0.18}, {UK3006, 0.2}, {BS1014, 0.1}, {BS1014, 0.12}, {BS1014, 0.14}, {UK3015, 0.1},
           {UK3015, 0.12}, {UK3015, 0.14}, {UK3015, 0.16}, {CK4016, 0.1}, {CK4016, 0.12},
           {CK4017, 0.1}, {CK4017, 0.12}, {CK4017, 0.14}, {BS1007, 0.1}, {BS1007, 0.12}, {PY2008, 0.1}}
```

This closes the connection.

```
In[198]:= CloseSQLConnection[conn]
```

It should be noted that `SQLArgument` is not supported in *Mathematica*-based queries.

Selecting Data with Raw SQL

The raw SQL command SELECT selects and returns data from a database. An alternative is to use the *Mathematica* command `SQLSelect`, described in "Selecting Data".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the ``DatabaseExamples` `` package, as described in "Using the Example Databases".

This loads *DatabaseLink* and connects to the *publisher* database.

```
In[199]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "publisher"];
```

This retrieves data within the table, *ROYSCHED*, for which the data in the *ROYALTY* column is between 0.11 and 0.12.

```
In[201]:= SQLExecute[conn,
           "SELECT * FROM ROYSCHED WHERE ROYALTY >= .11 AND ROYALTY <= .12"]
Out[201]= {{BS1011, 5001, 50000, 0.12}, {CP5018, 2001, 4000, 0.12},
           {BS1001, 1001, 5000, 0.12}, {PY2002, 1001, 5000, 0.12}, {PY2003, 2001, 5000, 0.12},
           {UK3004, 1001, 2000, 0.12}, {CK4005, 2001, 6000, 0.12}, {CP5010, 5001, 50000, 0.12},
           {PY2012, 5001, 50000, 0.12}, {PY2013, 5001, 50000, 0.12},
           {UK3006, 1001, 2000, 0.12}, {BS1014, 4001, 8000, 0.12}, {UK3015, 2001, 4000, 0.12},
           {CK4016, 5001, 15000, 0.12}, {CK4017, 2001, 8000, 0.12}, {BS1007, 5001, 50000, 0.12}}
```

This carries out the same SELECT statement but uses a prepared statement. The arguments to the statement are given as the third element of the `SQLExecute` command. The first argument is placed in the location of the `` `1` `` and the second in the location of the `` `2` ``.

```
In[202]:= SQLExecute[conn,
           "SELECT * FROM ROYSCHED WHERE ROYALTY >= `1` AND ROYALTY <= `2`", {0.11, 0.12}]
Out[202]= {{BS1011, 5001, 50 000, 0.12}, {CP5018, 2001, 4000, 0.12},
           {BS1001, 1001, 5000, 0.12}, {PY2002, 1001, 5000, 0.12}, {PY2003, 2001, 5000, 0.12},
           {UK3004, 1001, 2000, 0.12}, {CK4005, 2001, 6000, 0.12}, {CP5010, 5001, 50 000, 0.12},
           {PY2012, 5001, 50 000, 0.12}, {PY2013, 5001, 50 000, 0.12},
           {UK3006, 1001, 2000, 0.12}, {BS1014, 4001, 8000, 0.12}, {UK3015, 2001, 4000, 0.12},
           {CK4016, 5001, 15 000, 0.12}, {CK4017, 2001, 8000, 0.12}, {BS1007, 5001, 50 000, 0.12}}
```

Column and table names must be wrapped in `SQLColumn` and `SQLTable`, respectively. This will ensure they are not quoted as strings. The following selects elements of the *ROYALTY* column in the *ROYSCHED* table for which the *TITLE_ID* column value is `BS1011`.

```
In[203]:= SQLExecute[conn, "SELECT `1` FROM ROYSCHED WHERE TITLE_ID = `2`",
           {SQLColumn["ROYALTY"], "BS1011"}]
Out[203]= {{0.1}, {0.12}}
```

If you want to give a sequence of arguments to a prepared statement, you can use `SQLArgument`. This is described in Argument Sequences in SQL-Style Queries.

```
In[204]:= SQLExecute[conn, "SELECT `1` FROM ROYSCHED WHERE TITLE_ID = `2`",
           {SQLArgument[SQLColumn["LORANGE"],
             SQLColumn["HIRANGE"], SQLColumn["ROYALTY"]], "BS1011"}]
Out[204]= {{0, 5000, 0.1}, {5001, 50 000, 0.12}}
```

Many databases offer functions that apply to the results of a select operation. Typical examples are COUNT, MIN, MAX, SUM, and AVG. The documentation for your database will describe the details of the functions that are available. The following examples demonstrate some of these functions.

```
In[205]:= SQLExecute[conn, "SELECT COUNT(ROYALTY) FROM ROYSCHED"]
Out[205]= {{51}}
```

```
In[206]:= SQLExecute[conn, "SELECT MIN(ROYALTY) FROM ROYSCHED"]
Out[206]= {{0.1}}
```

Many databases allow you to apply mathematical functions such as +, -, *, or / to the results.

```
In[207]:= SQLExecute[conn, "SELECT ROYALTY * 2 FROM ROYSCHED"]
Out[207]= {{0.2}, {0.24}, {0.2}, {0.24}, {0.32}, {0.2}, {0.24}, {0.32}, {0.36}, {0.2}, {0.2}, {0.24}, {0.28},
           {0.2}, {0.24}, {0.28}, {0.2}, {0.24}, {0.28}, {0.36}, {0.4}, {0.2}, {0.24}, {0.32}, {0.32},
           {0.2}, {0.24}, {0.2}, {0.24}, {0.2}, {0.24}, {0.2}, {0.24}, {0.28}, {0.36}, {0.4}, {0.2}, {0.24},
           {0.28}, {0.2}, {0.24}, {0.28}, {0.32}, {0.2}, {0.24}, {0.2}, {0.24}, {0.28}, {0.2}, {0.24}, {0.2}}
```

```
In[208]:= SQLExecute[conn, "SELECT ROYALTY / 10 FROM ROYSCHED"]
Out[208]= {{0.01}, {0.012}, {0.01}, {0.012}, {0.016}, {0.01}, {0.012}, {0.016}, {0.018}, {0.01}, {0.01},
 {0.012}, {0.014}, {0.01}, {0.012}, {0.014}, {0.01}, {0.012}, {0.014}, {0.018}, {0.02},
 {0.01}, {0.012}, {0.016}, {0.016}, {0.01}, {0.012}, {0.01}, {0.012}, {0.01}, {0.012},
 {0.01}, {0.012}, {0.014}, {0.018}, {0.02}, {0.01}, {0.012}, {0.014}, {0.01}, {0.012},
 {0.014}, {0.016}, {0.01}, {0.012}, {0.01}, {0.012}, {0.014}, {0.01}, {0.012}, {0.01}}
```

```
In[209]:= SQLExecute[conn, "SELECT - ROYALTY FROM ROYSCHED"]
Out[209]= {{-0.1}, {-0.12}, {-0.1}, {-0.12}, {-0.16}, {-0.1}, {-0.12}, {-0.16}, {-0.18}, {-0.1}, {-0.1},
 {-0.12}, {-0.14}, {-0.1}, {-0.12}, {-0.14}, {-0.1}, {-0.12}, {-0.14}, {-0.18}, {-0.2},
 {-0.1}, {-0.12}, {-0.16}, {-0.16}, {-0.1}, {-0.12}, {-0.1}, {-0.12}, {-0.1}, {-0.12},
 {-0.1}, {-0.12}, {-0.14}, {-0.18}, {-0.2}, {-0.1}, {-0.12}, {-0.14}, {-0.1}, {-0.12},
 {-0.14}, {-0.16}, {-0.1}, {-0.12}, {-0.1}, {-0.12}, {-0.14}, {-0.1}, {-0.12}, {-0.1}}
```

You can also select only distinct values.

```
In[210]:= SQLExecute[conn, "SELECT DISTINCT ROYALTY FROM ROYSCHED"]
Out[210]= {{0.1}, {0.12}, {0.14}, {0.16}, {0.18}, {0.2}}
```

You can also group values.

```
In[211]:= SQLExecute[conn, "SELECT TITLE_ID,
      MIN(ROYALTY) FROM ROYSCHED GROUP BY TITLE_ID",
    "ShowColumnHeadings" → True] // TableForm
```

```
          TITLE_ID
          BS1011   0.1
          CP5018   0.1
          BS1001   0.1
          PS9999   0.1
          PY2002   0.1
          PY2003   0.1
          UK3004   0.1
          CK4005   0.1
Out[211]= CP5010   0.1
          PY2012   0.1
          PY2013   0.1
          UK3006   0.1
          BS1014   0.1
          UK3015   0.1
          CK4016   0.1
          CK4017   0.1
          BS1007   0.1
          PY2008   0.1
```

Many databases also support retrieving a range of results.

```
In[212]:= SQLExecute[conn, "SELECT TOP 5 * FROM ROYSCHED"]
Out[212]= {{BS1011, 0, 5000, 0.1}, {BS1011, 5001, 50 000, 0.12},
 {CP5018, 0, 2000, 0.1}, {CP5018, 2001, 4000, 0.12}, {CP5018, 4001, 50 000, 0.16}}
```

```
In[213]:= SQLExecute[conn, "SELECT LIMIT 5 10 * FROM ROYSCHED"]
Out[213]= {{BS1001, 0, 1000, 0.1}, {BS1001, 1001, 5000, 0.12},
 {BS1001, 5001, 7000, 0.16}, {BS1001, 7001, 50 000, 0.18},
 {PS9999, 0, 50 000, 0.1}, {PY2002, 0, 1000, 0.1}, {PY2002, 1001, 5000, 0.12},
 {PY2002, 5001, 50 000, 0.14}, {PY2003, 0, 2000, 0.1}, {PY2003, 2001, 5000, 0.12}}
```

More complex SELECT statements using INNER JOIN and OUTER JOIN can be used in a FROM clause to combine records from two tables.

```
In[214]:= SQLExecute[conn,
           "SELECT DISTINCT TITLES.TITLE FROM TITLES INNER JOIN ROYSCHED ON
             TITLES.TITLE_ID=ROYSCHED.TITLE_ID WHERE
             TITLES.PUB_ID='0877' AND ROYSCHED.ROYALTY > .1"]

Out[214]= {{Hamburger Again!}, {How to Burn a Compact Disk}, {Let Them Eat Cake!},
           {Made to Wonder: Cooking the Macabre}, {Too Many Cooks}, {Treasures of the Sierra Madre}}
```

```
In[215]:= SQLExecute[conn,
            "SELECT T.TITLE, T.TITLE_ID, MIN(R.ROYALTY) FROM ROYSCHED R, TITLES T LEFT
          OUTER JOIN ROYSCHED ON T.TITLE_ID = R.TITLE_ID GROUP BY T.TITLE, T.TITLE_ID
          ORDER BY R.ROYALTY, T.TITLE DESC", "ShowColumnHeadings" → True] // TableForm

            JDBC::error : Not in aggregate function or group by … E_ID
            ORDER BY R.ROYALTY, T.TITLE DESC]

Out[215]= $Failed
```

This closes the connection.

```
In[216]:= CloseSQLConnection[conn]
```

Creating Tables with Raw SQL

The raw SQL command CREATE TABLE creates tables in a database. An alternative is to use the *Mathematica* command `SQLCreateTable`, described in "Creating Tables".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

When creating a table, the result of `SQLExecute` is an integer specifying the number of rows affected by the query. If the table is created correctly, this integer will always be zero as no rows are affected when creating a new table.

Here is an example that creates a table. This loads *DatabaseLink* and connects to the *demo* database.

```
In[217]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

When a table is created, options can be given to restrict how data is stored within the database. In the following, a table is created with four columns. The *USERNAME* is a string-based column that cannot be `Null` and is the primary key. (A primary key is important to a table as it uniquely identifies a row within the table.) The other three columns (*ADDRESS*, *CITY*, and *ZIPCODE*) are regular string-based columns. However, they must be unique among all rows.

```
In[219]:= SQLExecute[conn,
           "CREATE TABLE ADDRESSES (
              USERNAME VARCHAR NOT NULL PRIMARY KEY,
              ADDRESS VARCHAR,
              CITY VARCHAR,
              ZIPCODE VARCHAR,
              UNIQUE (ADDRESS, CITY, ZIPCODE))"]
Out[219]= 0
```

In this example, a table with three columns is created. The first column is an integer that is an identity. This means that it is the primary key for the table and its value will be automatically incremented in each row. In other words, the value is not required when data is inserted; instead, the value will be the next available increment. The *USERNAME* is a string-based column that is the foreign key to the *ADDRESSES* table. The third column is a bit that has a default of 1 (i.e. if a value is not supplied when data is inserted it will be set to 1).

```
In[220]:= SQLExecute[conn, "CREATE TABLE MAILER (
            MAILERID INT IDENTITY,
            USERNAME VARCHAR NOT NULL,
            SENDMAILER BIT DEFAULT '1' NOT NULL,
            FOREIGN KEY (USERNAME) REFERENCES ADDRESSES (USERNAME))"]
Out[220]= 0
```

`SQLTableNames` verifies the tables exist in the database.

```
In[221]:= SQLTableNames[conn, "ADDRESSES"]
Out[221]= {ADDRESSES}
```

```
In[222]:= SQLTableNames[conn, "MAILER"]
Out[222]= {MAILER}
```

`SQLColumnNames` verifies the columns were created in the database.

```
In[223]:= SQLColumnNames[conn, "ADDRESSES"]
Out[223]= {{ADDRESSES, USERNAME}, {ADDRESSES, ADDRESS}, {ADDRESSES, CITY}, {ADDRESSES, ZIPCODE}}
```

```
In[224]:= SQLColumnNames[conn, "MAILER"]
Out[224]= {{MAILER, MAILERID}, {MAILER, USERNAME}, {MAILER, SENDMAILER}}
```

This deletes the tables and closes the connection.

```
In[225]:= SQLExecute[conn, "DROP TABLE MAILER"];
          SQLExecute[conn, "DROP TABLE ADDRESSES"];
          CloseSQLConnection[conn]
```

Other options may be available to you when creating tables depending on the database being used. See your database documentation for information on what options are specifically available.

Inserting Data with Raw SQL

The SQL command INSERT inserts data into a database. An alternative is to use the *Mathematica* command `SQLInsert`, as described in "Inserting Data".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

When inserting data, the result of `SQLExecute` is an integer specifying the number of rows affected by the query.

Here is an example that inserts data. This loads *DatabaseLink* and connects to the *demo* database.

```
In[228]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

As discussed in "Creating Tables with Raw SQL", the *ADDRESSES* and *MAILER* tables should be created.

```
In[230]:= SQLExecute[conn,
            "CREATE TABLE ADDRESSES (
              USERNAME VARCHAR NOT NULL PRIMARY KEY,
              ADDRESS VARCHAR,
              CITY VARCHAR,
              ZIPCODE VARCHAR,
              UNIQUE (ADDRESS, CITY, ZIPCODE))"];
          SQLExecute[conn, "CREATE TABLE MAILER (
            MAILERID INT IDENTITY,
            USERNAME VARCHAR NOT NULL,
            SENDMAILER BIT DEFAULT '1' NOT NULL,
            FOREIGN KEY (USERNAME) REFERENCES ADDRESSES (USERNAME))"];
```

This demonstrates an SQL statement that inserts a row into the *ADDRESSES* table.

```
In[232]:= SQLExecute[conn,
            "INSERT INTO ADDRESSES (USERNAME, ADDRESS, CITY, ZIPCODE) VALUES
               ('user1', '100 Trade Center', 'Champaign, IL', '61820')"]
Out[232]= 1
```

A SELECT statement verifies that the data has been added to the table.

```
In[233]:= SQLExecute[conn, "SELECT * FROM ADDRESSES"]
Out[233]= {{user1, 100 Trade Center, Champaign, IL, 61820}}
```

The *USERNAME* column is made to be a primary key, which means that it must be unique. If you try to insert the same data again, there is an error and the result is `$Failed`.

```
In[234]:= SQLExecute[conn,
            "INSERT INTO ADDRESSES (USERNAME, ADDRESS, CITY, ZIPCODE) VALUES
               ('user1', '100 Trade Center', 'Champaign, IL', '61820')"]

          JDBC::error: JDBC error: Unique constraint violation:  in st… Center', 'Champaign, IL', '61820')]

Out[234]= $Failed
```

With this command, the *USERNAME* parameter is unique, but *ADDRESS*, *CITY*, and *ZIPCODE* are not. These must also be unique and again there is an error.

```
In[235]:= SQLExecute[conn,
            "INSERT INTO ADDRESSES (USERNAME, ADDRESS, CITY, ZIPCODE) VALUES
               ('user2', '100 Trade Center', 'Champaign, IL', '61820')"]

          JDBC::error: JDBC error: Unique constraint violation: SYS_CT… Center', 'Champaign, IL', '61820')]

Out[235]= $Failed
```

This inserts unique values of *ADDRESS*, *CITY*, and *ZIPCODE*.

```
In[236]:= SQLExecute[conn,
            "INSERT INTO ADDRESSES (USERNAME, ADDRESS, CITY, ZIPCODE) VALUES
               ('user2', '200 Trade Center', 'Champaign, IL', '61820')"]
Out[236]= 1
```

A SELECT statement verifies that the data has been added to the table.

```
In[237]:= SQLExecute[conn, "SELECT * FROM ADDRESSES"]

Out[237]= {{user1, 100 Trade Center, Champaign, IL, 61820},
           {user2, 200 Trade Center, Champaign, IL, 61820}}
```

A prepared statement may be more useful for working with data to insert. In addition, `SQLArgument` may be useful to reduce the number of argument fields in the prepared statement. `SQLArgument` is described in Argument Sequences in SQL-Style Queries.

```
In[238]:= SQLExecute[conn,
            "INSERT INTO ADDRESSES (USERNAME, ADDRESS, CITY, ZIPCODE) VALUES
               (`1`)",
           {SQLArgument["user3", "300 Trade Center", "Champaign, IL", "61820"]}]
Out[238]= 1
```

A SELECT statement verifies that the data has been added to the table.

```
In[239]:= SQLExecute[conn, "SELECT * FROM ADDRESSES"]

Out[239]= {{user1, 100 Trade Center, Champaign, IL, 61820},
           {user2, 200 Trade Center, Champaign, IL, 61820},
           {user3, 300 Trade Center, Champaign, IL, 61820}}
```

Identity columns are very useful as they automatically increment their values and do not require a value. They are also the primary key for the table, which means they uniquely identify a row. Identity values should be set to `Null` in a SQL statement.

```
In[240]:= SQLColumnNames[conn, "MAILER"]

Out[240]= {{MAILER, MAILERID}, {MAILER, USERNAME}, {MAILER, SENDMAILER}}
```

```
In[241]:= SQLExecute[conn, "INSERT INTO MAILER
              (MAILERID, USERNAME, SENDMAILER) VALUES (NULL, 'user1', 0)"]

Out[241]= 1
```

A SELECT statement verifies that the data has been added to the table.

```
In[242]:= SQLExecute[conn, "SELECT * FROM MAILER"]

Out[242]= {{0, user1, False}}
```

Since *USERNAME* is a foreign key, its value must be present in *ADDRESSES*. The following fails because *user4* is not present in *ADDRESSES*.

```
In[243]:= SQLExecute[conn, "INSERT INTO MAILER
              (MAILERID, USERNAME, SENDMAILER) VALUES (NULL, 'user4', 0)"]

              JDBC::error: JDBC error: Integrity constraint violation – no … NDMAILER) VALUES (NULL, 'user4', 0)]

Out[243]= $Failed
```

The *SENDMAILER* column has a default value and is therefore not required when data is inserted.

```
In[244]:= SQLExecute[conn,
           "INSERT INTO MAILER (MAILERID, USERNAME) VALUES (NULL, 'user2')"]

Out[244]= 1
```

A SELECT statement verifies that the data exists in the database and ties the values together.

```
In[245]:= SQLExecute[conn,
            "SELECT USERNAME, ADDRESS, CITY, ZIPCODE, SENDMAILER FROM ADDRESSES,
              MAILER WHERE ADDRESSES.USERNAME = MAILER.USERNAME",
            "ShowColumnHeadings" → True] // TableForm

          USERNAME   ADDRESS            CITY            ZIPCODE   SENDMAILER
Out[245]= user1      100 Trade Center   Champaign, IL   61820     False
          user2      200 Trade Center   Champaign, IL   61820     False
```

This deletes the tables and closes the connection.

```
In[246]:= SQLExecute[conn, "DROP TABLE MAILER"];
          SQLExecute[conn, "DROP TABLE ADDRESSES"];
          CloseSQLConnection[conn]
```

Updating Data with Raw SQL

The raw SQL command UPDATE updates data in a database. An alternative is to use the *Mathematica* command `SQLUpdate`, described in "Updating Data".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

When updating data, the result of `SQLExecute` is an integer specifying the number of rows affected by the query.

Here is an example that updates data. This loads *DatabaseLink* and connects to the *demo* database.

```
In[249]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

As discussed in "Creating Tables with Raw SQL", the ADDRESSES and MAILER tables should be created.

```
In[251]:= SQLExecute[conn,
            "CREATE TABLE ADDRESSES (
              USERNAME VARCHAR NOT NULL PRIMARY KEY,
              ADDRESS VARCHAR,
              CITY VARCHAR,
              ZIPCODE VARCHAR,
              UNIQUE (ADDRESS, CITY, ZIPCODE))"];
          SQLExecute[conn, "CREATE TABLE MAILER (
            MAILERID INT IDENTITY,
            USERNAME VARCHAR NOT NULL,
            SENDMAILER BIT DEFAULT '1' NOT NULL,
            FOREIGN KEY (USERNAME) REFERENCES ADDRESSES (USERNAME))"];
          SQLExecute[conn,
            "INSERT INTO ADDRESSES (USERNAME, ADDRESS, CITY, ZIPCODE) VALUES (`1`)",
            {{SQLArgument["user1", "100 Trade Center", "Champaign, IL", "61820"]},
             {SQLArgument["user2", "200 Trade Center", "Champaign, IL", "61820"]},
             {SQLArgument["user3", "300 Trade Center", "Champaign, IL", "61820"]}}
           ];
          SQLExecute[conn,
            "INSERT INTO MAILER (MAILERID, USERNAME, SENDMAILER) VALUES (`1`)",
            {{SQLArgument[Null, "user1", False]},
             {SQLArgument[Null, "user2", False]}}
           ];
```

This executes an SQL statement that updates a row in the MAILER table. This query updates the SENDMAILER column based on the value of USERNAME. Many update statements may be created using conditions that work with values in columns.

```
In[255]:= SQLExecute[conn,
            "UPDATE MAILER SET SENDMAILER = 1 WHERE USERNAME = 'user1'"]

Out[255]= 1
```

A SELECT statement verifies that the data has been changed in the table.

```
In[256]:= SQLExecute[conn, "SELECT * FROM MAILER"]

Out[256]= {{0, user1, True}, {1, user2, False}}
```

Using prepared statements, you can dynamically create SQL statements that update data within the database. You can combine this with a simple *Mathematica* function. This example updates the address for a particular user.

```
In[257]:= SetAddress[username_String, address_String] :=
           SQLExecute[conn,
             "UPDATE ADDRESSES SET ADDRESS = `2` WHERE USERNAME = `1`",
            {username, address}]

In[258]:= SetAddress["user1", "100 Trade Center Office 123"]

Out[258]= 1
```

A SELECT statement verifies that the data has been changed in the table.

```
In[259]:= SQLExecute[conn, "SELECT * FROM ADDRESSES"]

Out[259]= {{user1, 100 Trade Center Office 123, Champaign, IL, 61820},
           {user2, 200 Trade Center, Champaign, IL, 61820},
           {user3, 300 Trade Center, Champaign, IL, 61820}}
```

The same restrictions that apply to inserts also apply to updates. Thus, if you try to update an ADDRESS value to equal the ADDRESS value of another row, an error will be returned; this table requires them to be unique.

```
In[260]:= SetAddress["user1", "200 Trade Center"]

          JDBC::error : JDBC error: Unique constraint violation: SYS_CT … SET ADDRESS = ? WHERE USERNAME = ?]

Out[260]= $Failed
```

This deletes the tables and closes the connection.

```
In[261]:= SQLExecute[conn, "DROP TABLE MAILER"];
          SQLExecute[conn, "DROP TABLE ADDRESSES"];
          CloseSQLConnection[conn]
```

Deleting Data with Raw SQL

The raw SQL command DELETE deletes data from a database. An alternative is to use the *Mathematica* command `SQLDelete`, described in "Deleting Data".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

When deleting data, the result of `SQLExecute` is an integer specifying the number of rows affected by the query.

Here is an example that removes data. This loads *DatabaseLink* and connects to the *demo* database.

```
In[264]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

As discussed in "Creating Tables with Raw SQL", the *ADDRESSES* and *MAILER* tables should be created.

```
In[266]:= SQLExecute[conn,
            "CREATE TABLE ADDRESSES (
              USERNAME VARCHAR NOT NULL PRIMARY KEY,
              ADDRESS VARCHAR,
              CITY VARCHAR,
              ZIPCODE VARCHAR,
              UNIQUE (ADDRESS, CITY, ZIPCODE))"];
          SQLExecute[conn, "CREATE TABLE MAILER (
            MAILERID INT IDENTITY,
            USERNAME VARCHAR NOT NULL,
            SENDMAILER BIT DEFAULT '1' NOT NULL,
            FOREIGN KEY (USERNAME) REFERENCES ADDRESSES (USERNAME))"];
          SQLExecute[conn,
            "INSERT INTO ADDRESSES (USERNAME, ADDRESS, CITY, ZIPCODE) VALUES (`1`)",
            {{SQLArgument["user1", "100 Trade Center", "Champaign, IL", "61820"]},
             {SQLArgument["user2", "200 Trade Center", "Champaign, IL", "61820"]},
             {SQLArgument["user3", "300 Trade Center", "Champaign, IL", "61820"]}}
           ];
          SQLExecute[conn,
            "INSERT INTO MAILER (MAILERID, USERNAME, SENDMAILER) VALUES (`1`)",
            {{SQLArgument[Null, "user1", False]},
             {SQLArgument[Null, "user2", True]}}
           ];
```

Here are the contents of the *ADDRESSES* table.

```
In[270]:= SQLExecute[conn, "SELECT * FROM ADDRESSES"]

Out[270]= {{user1, 100 Trade Center, Champaign, IL, 61820},
           {user2, 200 Trade Center, Champaign, IL, 61820},
           {user3, 300 Trade Center, Champaign, IL, 61820}}
```

Here are the contents of the *MAILER* table.

```
In[271]:= SQLExecute[conn, "SELECT * FROM MAILER"]
Out[271]= {{0, user1, False}, {1, user2, True}}
```

This executes an SQL statement that deletes a row in the *MAILER* table. It deletes any rows for which the value in the *SENDMAILER* column is 0 (or `False`). Delete statements can be created using conditions that depend on the values in columns. Since one row has been deleted, the result is 1.

```
In[272]:= SQLExecute[conn,
            "DELETE FROM MAILER WHERE SENDMAILER = 0"]
Out[272]= 1
```

A SELECT statement verifies that the data has been changed in the table.

```
In[273]:= SQLExecute[conn, "SELECT * FROM MAILER"]
Out[273]= {{1, user2, True}}
```

Using prepared statements, you can dynamically create SQL statements that delete data within the database. You can combine this with a simple *Mathematica* function. This example deletes an address for a particular user.

```
In[274]:= DeleteAddress[username_String] :=
           SQLExecute[conn,
             "DELETE FROM ADDRESSES WHERE USERNAME = `1`", {username}]

In[275]:= DeleteAddress["user3"]
Out[275]= 1
```

A SELECT statement verifies that the data has been changed in the table.

```
In[276]:= SQLExecute[conn, "SELECT * FROM ADDRESSES"]
Out[276]= {{user1, 100 Trade Center, Champaign, IL, 61820},
           {user2, 200 Trade Center, Champaign, IL, 61820}}
```

Any restrictions on the values in tables also apply when data is deleted. Thus, if you try to delete an *ADDRESS* value that is referenced by an item in the *MAILER* table, an error occurs.

```
In[277]:= DeleteAddress["user2"]

          JDBC::error: JDBC error: Integrity constraint violation SYS_ … E FROM ADDRESSES WHERE USERNAME = ?]

Out[277]= $Failed
```

This deletes the tables and closes the connection.

```
In[278]:= SQLExecute[conn, "DROP TABLE MAILER"];
          SQLExecute[conn, "DROP TABLE ADDRESSES"];
          CloseSQLConnection[conn]
```

Dropping Tables with Raw SQL

The raw SQL command DROP TABLE drops tables from a database. An alternative is to use the *Mathematica* command `SQLDropTable`, described in "Dropping Tables".

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

When dropping a table, the result of `SQLExecute` will be `$Failed` if there is an error.

Here is an example that drops a table. This loads *DatabaseLink* and connects to the *demo* database.

```
In[281]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

A simple table is created and two rows are inserted.

```
In[283]:= SQLExecute[conn,
            "CREATE TABLE TEST (COL1 INTEGER, COL2 INTEGER)"];
          SQLExecute[conn, "INSERT INTO TEST (COL1, COL2) VALUES (`1`)",
            {{SQLArgument[5, 6]}, {SQLArgument[7, 9]}}
           ];
          SQLExecute[conn, "SELECT * FROM TEST"]
Out[285]= {{5, 6}, {7, 9}}
```

An SQL statement that drops the *TEST* table is executed.

```
In[286]:= SQLExecute[conn, "DROP TABLE TEST"]
Out[286]= 0
```

This confirms that the *TEST* table is no longer in the database.

```
In[287]:= SQLTableNames[conn, "TEST"]
Out[287]= {}
```

This closes the connection.

```
In[288]:= CloseSQLConnection[conn]
```

It should be noted that it is not permitted to drop a table that is referenced by another.

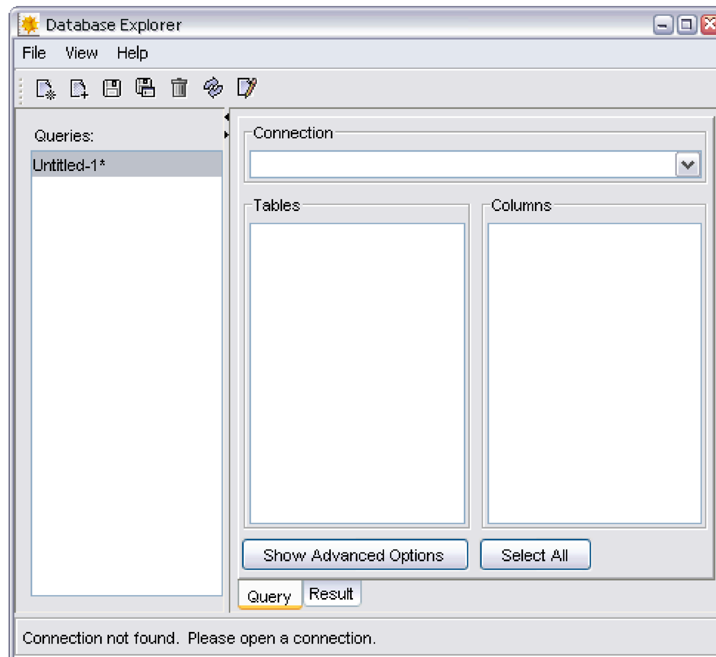
The Database Explorer

The Database Explorer is a graphical interface to *DatabaseLink*. It provides a number of useful functions, such as managing connections and working with the data in a database. It can be launched by loading *DatabaseLink* and executing the command `DatabaseExplorer`.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

```
In[289]:= Needs["DatabaseLink`"];
          DatabaseExplorer[]

Out[290]= - GUIObject -
```

On Windows it appears as follows.

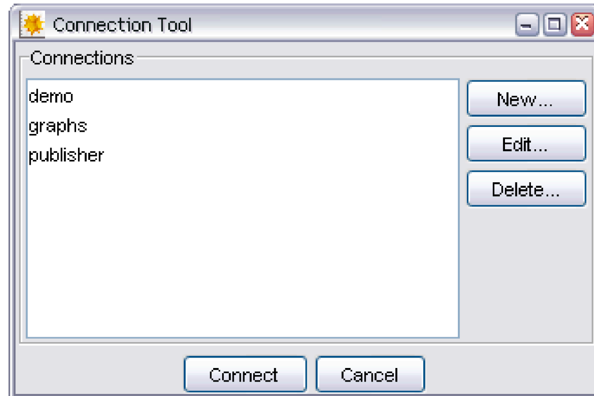


At this point you can connect to a database and make queries from its tables. When you have selected the data, it can be used to create a report as a *Mathematica* notebook. This will allow you to work with the results in *Mathematica*.

This version of the Database Explorer can only select and read data from a database.

The Connection Tool

From the main **Database Explorer** window, you can open the **Connection Tool** by using the **Connect to a data source** button. It can also be launched by executing the command `OpenSQLConnection`, described in Database Connections: Establishing a Connection.



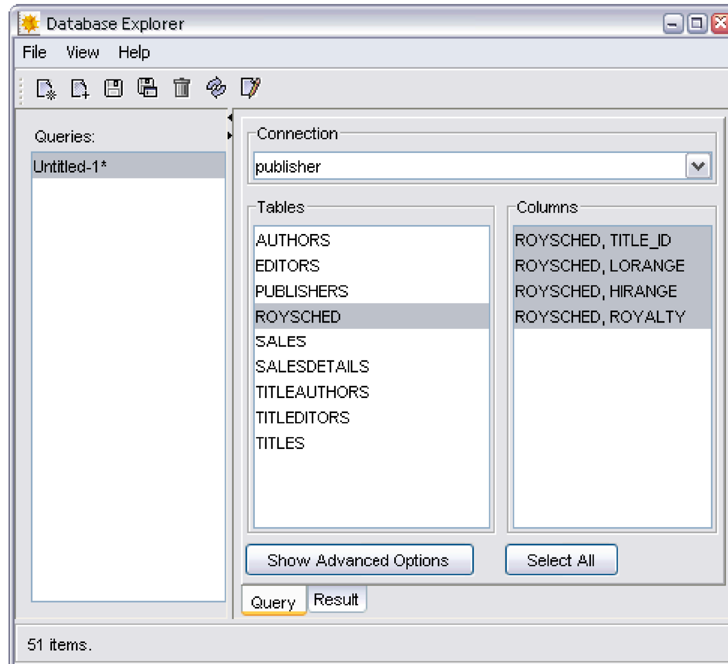
If you do not see the sample databases shown in the picture, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

The **Connection Tool** shows all of the database connections that appear in configuration files in `DatabaseResources` directories. The details of named connections and their configuration files are described in "Database Resources". From the **Connection Tool** you can select a connection and edit or delete it. You can also create a new connection, as described in "New Connection Wizard".

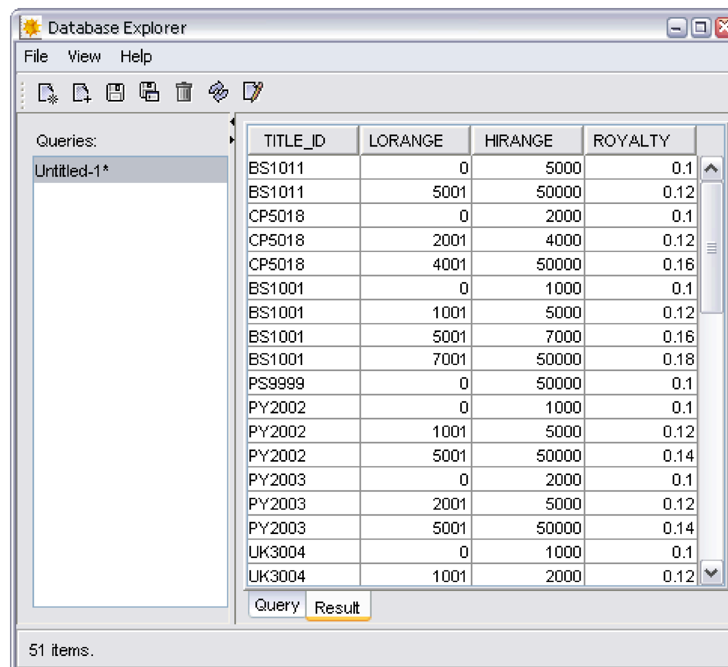
You can use the **Connect** button to open a connection to the database that was selected and update the main **Database Explorer** window. You can now make queries from the database.

Querying the Database

When you have connected to a database, as described previously, the **Database Explorer** shows the actual database in the **Connection** list and the tables in a scroll list. The following picture shows the result of connecting to the *publisher* database. This contains a number of tables. If you select one of the tables, its columns will be shown, and you can select any of them. A button for selecting all the columns is also provided.

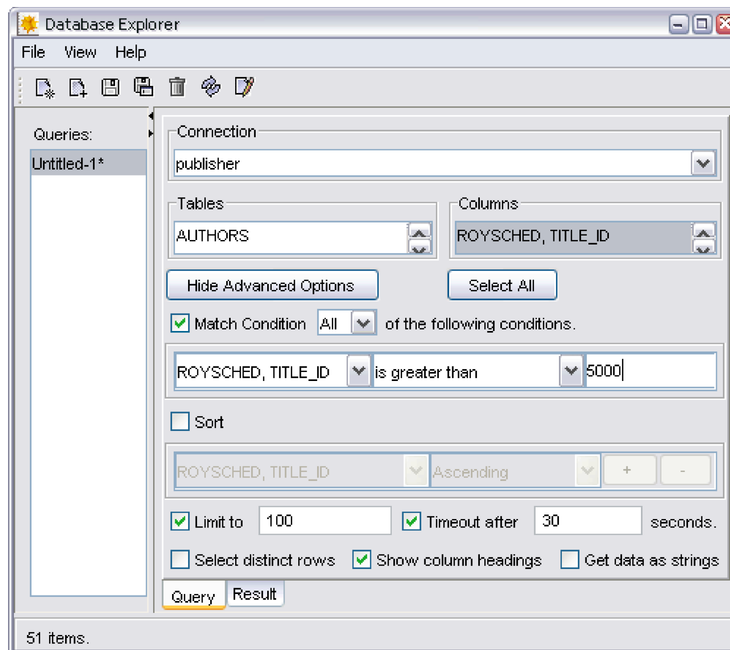


Clicking the **Result** tab selects the data from the table and selected columns. Here is an example.



TITLE_ID	LORANGE	HIRANGE	ROYALTY
BS1011	0	5000	0.1
BS1011	5001	50000	0.12
CP5018	0	2000	0.1
CP5018	2001	4000	0.12
CP5018	4001	50000	0.16
BS1001	0	1000	0.1
BS1001	1001	5000	0.12
BS1001	5001	7000	0.16
BS1001	7001	50000	0.18
PS9999	0	50000	0.1
PY2002	0	1000	0.1
PY2002	1001	5000	0.12
PY2002	5001	50000	0.14
PY2003	0	2000	0.1
PY2003	2001	5000	0.12
PY2003	5001	50000	0.14
UK3004	0	1000	0.1
UK3004	1001	2000	0.12

The **Query** page also has an **Advanced Options** button. When you click this button, more options for forming the query are provided. For example, you can put various conditions on columns. Here is an example where data in the TITLE_ID column must be greater than 5000.



Clicking the **Result** tab will run the query and display the results.

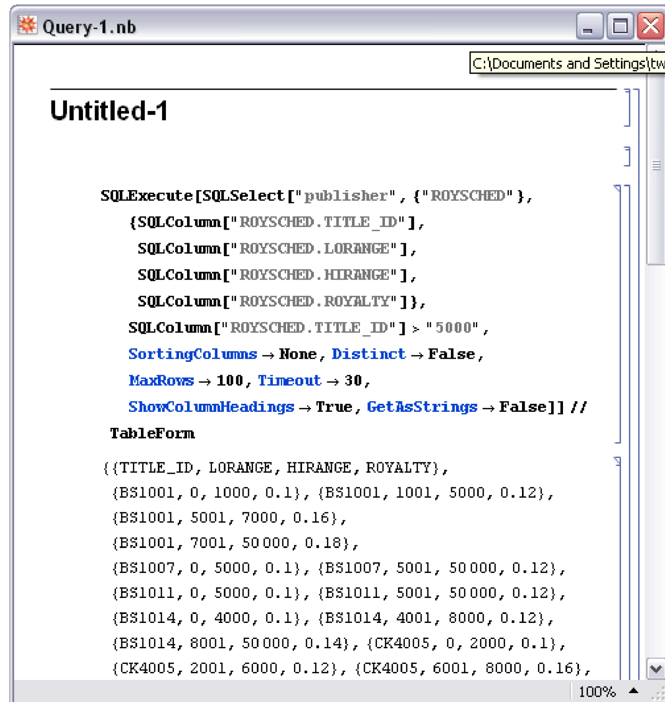
Saving Queries

When you have set up a query, it can be saved with the **Save the Query** button. When you click this button a **Save File** dialog box appears that includes a number of locations in `DatabaseResources` directories. (`DatabaseResources` directories are described in "Database Resources".) When you launch the **Database Explorer**, all the queries that have been saved are made available and can be run.

Exporting to *Mathematica*

When you have set up a query, the data can be extracted into a report in a *Mathematica* notebook document. This can be used for printing or for further work in *Mathematica.*

To generate a report, click the **Create a notebook** button. Here is a sample notebook.



When the data is in *Mathematica*, you can process it further with all the tools that *Mathematica* provides.

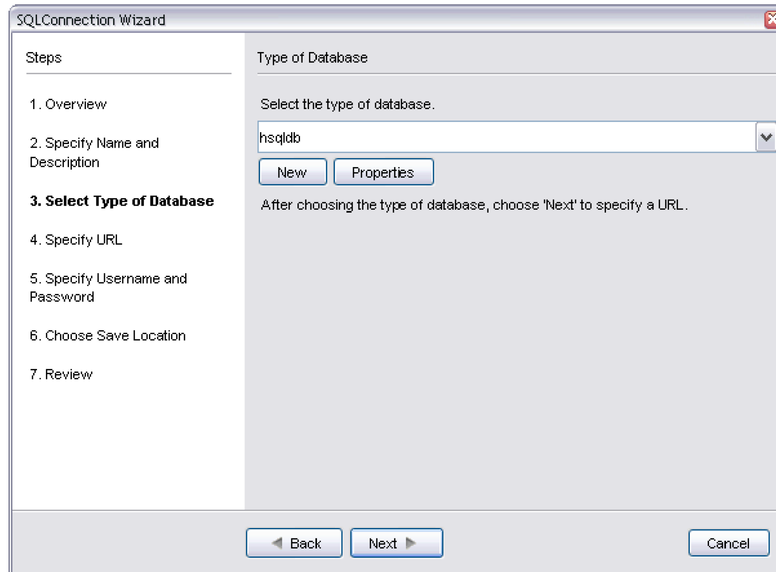
New Connection Wizard

The **New Connection Wizard** is available from the **Connection Tool**, which can be launched either from the **Database Explorer** (described previously) or by executing the command `OpenSQLConnection[]` (described in "Database Connections: Establishing a Connection"). It will create a new named connection that will be available for future uses. The information about the connection will be written in a configuration file as described in "Database Resources".

The wizard takes you through the following steps for creating a new connection.

1. Overview.
2. Specify name and description.
3. Select type of database.
4. Specify URL.
5. Specify username and password.
6. Choose save location.
7. Review.

A view of the third step is shown in the following picture. In this step, the type of the database is selected from a list. There is also a button for entering a new type of database. The list includes types that have been installed in the system as described in Database Resources: JDBC Configuration.



Each screen of the wizard has a full description. When it has finished, a new named connection has been created. This can be used by `OpenSQLConnection` and will show up in the **Connection Tool**.

Advanced Topics

Data Type Mapping

One of the most important issues for using a database is the conversion of data as it is stored and retrieved from a database. This tutorial will discuss how *Mathematica* expressions interact with data stored in a database.

The following table shows the mappings between data types and *Mathematica* expressions. For example, a *Mathematica* `Integer` expression can be stored in SQL integral types such as INTEGER and TINYINT. In addition, if data from a column that is of type VARCHAR is selected, this will result in a *Mathematica* `String` expression.

Mathematica expression	*data type*
`String`	used mostly with SQL types such as CHAR, VARCHAR, and LONGVARCHAR
`Integer`	used mostly with SQL types such as INTEGER, TINYINT, SMALLINT, and BIGINT
`Real`	used mostly with SQL types such as DOUBLE, FLOAT, and REAL
`True`	used mostly with the SQL type BIT
`False`	used mostly with the SQL type BIT
`Null`	used mostly with the SQL type NULL
`SQLBinary`	used mostly with SQL types such as BINARY, VARBINARY, and LONGVARBINARY
`SQLDateTime`	used mostly with SQL types such as DATE, TIME, and TIMESTAMP
`SQLExpr`	a special type of binary data that is used to store *Mathematica* expressions

The mapping between *Mathematica* expressions and data types stored in a database.

Atomic *Mathematica* expressions such as `String`, `Integer`, `Real`, `True`, `False`, and `Null`, and compound expressions formed from `SQLBinary`, `SQLDateTime`, and `SQLExpr` are converted to and from Java objects. These Java objects are then processed with JDBC operations taking

String Integer Real True False Null

SQLBinary SQLDateTime SQLExpr

advantage of any encoding or escaping functionality that is provided by the JDBC driver. It is typical that they contain code specific to a database for encoding a value passed into or received from a query. Since these drivers are often implemented by the makers of the database, it is very advantageous to use their functionality as much as possible.

Certain data types require *Mathematica* expressions that use a special wrapper. For example, the data type BINARY requires a *Mathematica* expression that uses the wrapper `SQLBinary`. These wrappers are necessary to prevent ambiguities in the command structure.

SQLBinary

`SQLBinary` can be used to work with binary data in a database. This allows you to store data such as images or compiled code.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

This loads *DatabaseLink* and connects to the *demo* database.

```
In[291]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

This generates a string that contains a GIF image.

```
In[293]:= gif = ExportString[Plot[Sin[x], {x, 0, 2 Pi}], "GIF"]
```


```
Out[293]= GIF89ahÔÄ!!!!!!
   (((000888@@@HHHPPPXXX```hhhpppxxx>>>>>>---⇕⇕⇕■■■§§§¯¯¯···¿¿¿ÇÇÇÏÏÏ×××ßßßçççïïï÷÷÷ÿÿÿ,hÔ!ÿà
   '~di@hª®lë¾p,Ïtmßx®ï|ïÿÀ pH,!E-£rÉl :■P
   Y⟷V£Ø¬vËÅRGÔïGÜ-:=Ïè4=!⟺ªßð¸ \ÉFJ§Ö9êòh!!! !z_×hucn=$! !!!<< !!!! ~! !_¤¥K,bd_ !!
>> .!
!! !¦¸¹ <xvc,_!!!º3 !!!!⟶ºÌÍ ?ªp É!9 !!! ÎÚÛ6Ði!!
Ë;
!!Üéê +Þf!!!A!!äëøêí ]!! E Îå!Èmß!!¨!á- Á-■!s !ÄÒá \&⟸ FÜ¨_⇕⟶
\!=1Á` !Ç:z &ÿ>±p` ⟹
!( ◇ùFe=    î=ñ !Í■el .„ºà¡⟺!!- ⋆Õ"ôÈ!!f
   ,XJõIÓ"!! ⟶0µª×⌗W>>dUÃõ«Ù!a□,}Sö¬Û!i ¬-! !éÛ»8âö⇕uN!■x !ËðèÄ !■D
d
^ìB¯!
\!,Îñ⇕ ãË)!ãØP @ç◇!!äa!í¨×>>⟸¥8 {&½X³„Ä¸. ⇕Ìºµ=
tñ„Ùõ,!!⟷ýåÍØw
\⟵!sy8-:¸`ã /⟺7w¦Ú9^è.!üÛV !:õ»\>>ìN7üûYì+ Óv&Ý¼Yô)<!p©®ºûªðQh !Øý >þ"ý T@ @åW!!!$>>Ïÿi & ø=$
   `!D!!º@□ A(Âx!Í- !G!jPÀ-ù0øáI !¶>>R-$@¨O! =xx⟸,.âc H@⟶Xã@ùéø \ ;æ=_]TÑ!¤6ð×Ø "G>!¹
|!_¤ ⇕N¦!
,„ðÀaUFV¹H/`⟵!!¬⟺c¢yY ⟵Hü⟸C!!¸e⟵⟷p´ !fi8!è !lt¦aç !`àáX!! (ø!⟶fö⟶Å■x !Ç- !:m©h !0 & ^⟺7⇕´X
NÚQ!◇ .<<£§■Þ )^!⟸ :>>f!z⇕!!¸ªª■7¤º⟵@³ÖiÃ¨£I⟵k !~ !0ÝhJþº &
 (À=Æ!-¥ !ºÊº !¡Ò6ë !ÿ
Ì~vªµLÍP¬s¡rKé
\p~§©¸\\-«så¢Ûm
```

```
\á~ÇÁKº!V
\¸ù×¥½tÀ@«Ñò!ESÙZ ·®Àýº°õò! /Ât<、B»!ò
ñ½-◇û!Å! !aÓÃ ! ~Û1 *µåbÁ¤ÃÅBÀ5"⟷2Z,ìK`¾/!aÐ¾
»ÇrÍ>!∵2 ≪ !ó◇Ã >! !     ²Ðy©Àñ„À!½、Ä!4MçÒN ×ÐÎÑGRYu7(!íäÏ [·ÍÎtâ!¶Õ 'Ð<©! !⊇ =□*⟸J¶Û„⟶¶=jÓ]7
  f«ê⟷z¿@!Õ=z ø⊇¤ *Ö>>W„!á@JÝxf× !À -ä=cÊ8©==gÞ、áª>ê9ÿâ:=«Ú÷èvlkìÝ!Kü ,s;¡áÛéò]æÀµçN!£¾ç =ôî*ì=ßðÀ= 6
  ñ⟹÷\ü!Ï !Ï5¦Y¦R (óÔW«B ;Â;■üõÚ[!¨ ÷ß¤>!ËÙ□_§øÜ:= !æöè ·!þûð■ /?â©°
  `î¯¯>:=ñë■þþå!` ÷þç>⟷!0 «þC  ÿæ⇕ÀþAO:!dà ! %8 ÞÙOP !Ì  !7ÈÁ !zð□
\¡GHÂ ⟷ð∵ (L¡
WÈÂ R¯ ~Ð:=Ð¾□0! ^ï!ap  ÂJ ÷«ÎÅ@≪!⟸◇»x (¶⟹a.W  !S- (8&²àR%ð¡ÍÒåD±!∵_E |×¢&h!¥ÅÉ P=Ý¸V ºÿÁl¡=={Û¢
N>>!4 !Ï„⟶      £b·¾k„Ñ!  a ]Òà8@ÜEp!¾ {@!ÏxG  ¸ ¬„rl! !égÈÛé ===¬Ý"
  _■@!a⟸⟹ìâ ! ? Ç4F⟸==!å ) 9¾Xyl×⟵t◇ !éµ!¡⇔„Nûi¥Ð° !$ )¢MhItÒ ëX3a„J_ ;H¥{li´NÉ i» ¬Q.yY3„yI>ôQYÕ:9
  ³Ü\⟸⟹cã„èøÈ°Y²FÔ4äÖ¸ù!mJskÉªR8Å!6Z ~>>⟵¿;!0O⟹Ì¤¹ @Ö⟶góÜ¦µ!¹S⇕g [§¶úB2½⟶Sg¦ô!6SöÍï ⟹⟶ãL¨«
  !⟺³n !n⟶¤Á§E !çËûhÿ
\、„□çhþ!ÏÃÅò ;&ÃØá!J !rNqr
Å
\Û"99h:>>¥_◇!Fï¢ÑsNÎWµ⟶= ! ='R·ü-ML!¦ :ñw! "iÒs>>¼
\NO×ºWº¥§ûD]G! ¤Ô@]I>>⟶ëç«§¾! ^
  u¤xèI◇ZV6¨B©jr)U¦ú¾³ÞA⟹tÛ)M°Ê! ^0*:p⊇Ú!Ý¢WqÚµ:££ +Mä⟷! !úâ h-[¿":BFv|¨ùÊG!yÙ !lV )G-dg+[$_ö²6]
  ==X9ÛÙ¨!g )[Õ]kE VgÄt`!m \hiBÑ3!⟺T■ÝH⟹¾4Û  Ì  %=
JqK°Û„¬V¶Ëõ¬⟵DVâFw!ÿÇ- i{Ý!¨h
  ]¥UwG`! |ÄíXã!AaI Tô¦⇕=LÝ≪U©øÛ\}w!ÎTC }„U^öTWWï%ÁzåÀ×O!¸ !µECr |à92§!á⇔Ã ~-! ]ôWÂ
.:zpÑÜ⟹d¸! )}!■  1aq %! ! ·Õï>>MÐ^DD!  xMï }å  :===rwÅ"è@¢     >>
\!  Q!Æq      .◇≪
:=¢Äöò1⇕·°:⟺HDÈ'P²!⟷ÌcR
  ⟶_R>CE、yd(ïÁKv! ! !Àå .{9Ê;î!LÊlæ3:=`Í [¨P⇔qqe_„y°P`Iµ¬◇Û!w  m{Â! ! !æGÔ¹c !P-„yÐ !⟵¶  nf¹!þ [!
\ `B⇕~ô
D h "@ÀsvÆ¡Sæ!d6 !²、„4 、!_ã ! p@≪µjV»@!Ê  Ç5:Ífg1AÖÖ.Æ!°g !lÀ É(t⇕×!V`«  ! ! !- !ðü !
$`!Pöª Õlg¯ÀÚ@-!Vp$  ÓÎ4³!co⇕À !■ ! !!  -?- !¨-¶ëD !T8ÐÝ
ãÄ! ! @N := !þ⟺  ÀÁ=>>!Bpáî !5Ä 'Nñ==o¤! ;
```


`ToCharacterCode` is used to create a list of bytes that represent the image. This list will also be wrapped in `SQLBinary`.

```
In[294]:= byteData = SQLBinary[ToCharacterCode[gif]];
```

This creates a table for demonstration purposes.

```
In[295]:= SQLCreateTable[conn, "BINTABLE",
            {SQLColumn["BINCOL", "DataTypeName" -> "BINARY"]}]
Out[295]= 0
```

This inserts the data into the table.

```
In[296]:= SQLInsert[conn, "BINTABLE", {"BINCOL"}, {byteData}]
Out[296]= 1
```

The data is now retrieved using `SQLSelect`. Since it is binary data, it is returned as an `SQLBinary` expression.

```
In[297]:= data = SQLSelect[conn, "BINTABLE"];
```

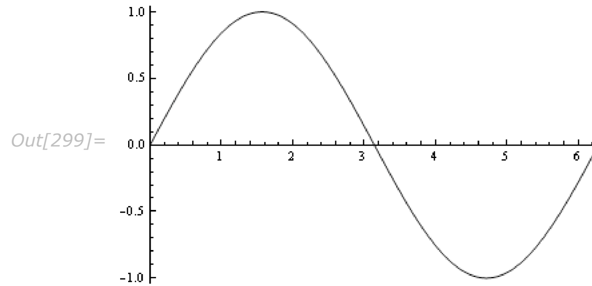
Then, the data is converted back into a string using `FromCharacterCode`.

```
In[298]:= gifData = FromCharacterCode[ data[[1, 1, 1]] ];
```

Finally, you can import the data and display it.

In[299]:=
```
Show[ImportString[gifData, "GIF"]]
```

Out[299]=



This drops the table and closes the connection.

In[300]:=
```
SQLDropTable[conn, "BINTABLE"];
CloseSQLConnection[conn];
```

SQLDateTime

`SQLDateTime` allows you to store and retrieve date and time information. It also allows you to execute queries that depend on specific dates or times.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

This loads *DatabaseLink* and connects to the *demo* database.

In[302]:=
```
Needs["DatabaseLink`"];
conn = OpenSQLConnection[ "demo"];
```

You can create a table for demonstration purposes. This table contains *DATE*, *TIME*, *DATETIME*, and *TIMESTAMP* columns.

In[304]:=
```
SQLCreateTable[conn, "DATETIMETABLE",
 {SQLColumn["DATECOL", "DataTypeName" -> "DATE"],
  SQLColumn["TIMECOL", "DataTypeName" -> "TIME"],
  SQLColumn["DATETIMECOL", "DataTypeName" -> "DATETIME"],
  SQLColumn["TIMESTAMPCOL", "DataTypeName" -> "TIMESTAMP"]}]
```

Out[304]=
```
0
```

Now, you can insert data into the table. You can use the output of the *Mathematica* `DateList[]` function for all data types except for the data type TIME; for this you must specify a list of three integers that specify hours, minutes and seconds. Note that DATE will only use the date information from `DateList[]` and not the time information. DATETIME and TIMESTAMP will use both and also nanoseconds.

```
In[305]:= SQLInsert[conn, "DATETIMETABLE",
           {"DATECOL", "TIMECOL", "DATETIMECOL", "TIMESTAMPCOL"},
           {SQLDateTime[DateList[]], SQLDateTime[{3, 4, 5}],
            SQLDateTime[DateList[]], SQLDateTime[DateList[]]}]
Out[305]= 1
```

SQLSelect can be used to retrieve the data from the database. The data will be returned as SQLDateTime expressions.

```
In[306]:= SQLSelect[conn, "DATETIMETABLE"]
Out[306]= {{SQLDateTime[{2006, 2, 7}], SQLDateTime[{3, 4, 5}],
            SQLDateTime[{2006, 2, 7, 14, 34, 58.3855}], SQLDateTime[{2006, 2, 7, 14, 34, 58.3855}]}}
```

This drops the table and closes the connection.

```
In[307]:= SQLDropTable[conn, "DATETIMETABLE"];
          CloseSQLConnection[conn];
```

SQLExpr

SQLExpr can be used to store *Mathematica* expressions in a database. When they are retrieved, they are converted back into *Mathematica* expressions.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the DatabaseExamples` package, as described in "Using the Example Databases".

This loads *DatabaseLink* and connects to the *demo* database.

```
In[309]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

In order to store a *Mathematica* expression, you need to create a column that can be used to store a string such as VARCHAR.

```
In[311]:= SQLCreateTable[conn, "EXPRTABLE",
           {SQLColumn["EXPRCOL", "DataTypeName" -> "VARCHAR"]}]
Out[311]= 0
```

This inserts a *Mathematica* expression into the database.

```
In[312]:= SQLInsert[conn, "EXPRTABLE", {"EXPRCOL"}, {SQLExpr[Sin[x^2]]}]
Out[312]= 1
```

`SQLSelect` retrieves the data from the database. The data is returned as an `SQLExpr` expression.

```
In[313]:= data = SQLSelect[conn, "EXPRTABLE"]
Out[313]= {{SQLExpr[Sin[x^2]]}}
```

This drops the table and closes the connection.

```
In[314]:= SQLDropTable[conn, "EXPRTABLE"];
          CloseSQLConnection[conn];
```

Result Sets

When many rows of data are returned from a database query, a significant amount of memory may be required to hold the result. If all of the data does not need to be available at the same time it might be preferable to get the result row by row or a few rows at a time. Rows can then be processed individually or in small groups. This functionality is provided by the SQL result set functions of *DatabaseLink*.

Basic Result Set Operations

Result set operations involve creating a result set, reading from it, and then closing it. This section discusses the basic ways to work with result sets.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

`SQLResultSetOpen[`*query*`]`	create an SQL result set based on *query*
`SQLResultSetOpen[`*query*`,`*opts*`]`	create an SQL result set using options *opts*
`SQLResultSetRead[`*rs*`]`	read a row from result set *rs*
`SQLResultSetRead[`*rs*`,`*num*`]`	read *num* rows from result set *rs*
`SQLResultSetClose[`*rs*`]`	close result set *rs*

Basic result set functions.

The query argument to `SQLResultSetOpen` is a function that selects data using either `SQLSelect` or `SQLExecute`. Here is an example.

First, the *DatabaseLink* package is loaded and a connection is made to the *publisher* example database.

```
In[316]:= << DatabaseLink`;
          conn = OpenSQLConnection["publisher"];
```

You can use this connection to read eight rows from the *ROYSCHED* table.

```
In[318]:= SQLExecute[conn, "SELECT * FROM roysched", "MaxRows" → 8] // TableForm
```

```
Out[318]= BS1011   0      5000     0.1
          BS1011   5001   50 000   0.12
          CP5018   0      2000     0.1
          CP5018   2001   4000     0.12
          CP5018   4001   50 000   0.16
          BS1001   0      1000     0.1
          BS1001   1001   5000     0.12
          BS1001   5001   7000     0.16
```

You can also obtain a result set from the same query.

```
In[319]:= rs = SQLResultSetOpen[SQLExecute[conn, "SELECT * FROM roysched", "MaxRows" → 8]]

Out[319]= SQLResultSet[0, <>, Scrollable]
```

`SQLResultSetRead` reads from the result set and returns the rows that were read. After reading from a result set, the next read will read the next row. The following example reads a single row. Since the result set was just created, it reads the first row.

```
In[320]:= SQLResultSetRead[rs]

Out[320]= {BS1011, 0, 5000, 0.1}
```

The following reads the second and third rows.

```
In[321]:= SQLResultSetRead[rs, 2]

Out[321]= {{BS1011, 5001, 50 000, 0.12}, {CP5018, 0, 2000, 0.1}}
```

By default `SQLResultSetRead` maps data types into various *Mathematica* expressions. However, setting the option `"GetAsStrings"` to `True` gets results as string expressions.

```
In[322]:= SQLResultSetRead[rs, "GetAsStrings" → True] // InputForm

Out[322]= {"CP5018", "2001", "4000", "0.12"}
```

If you want to process each row individually, you can use a construct like the following. It reads the remaining rows and sums the last element of each row. Since there were eight rows in the result set and four had already been read, this operation will read four rows. When `SQLResultSetRead` returns something that is not a list you have reached the end of the result set.

```
In[323]:= res = 0; While[ListQ[ data = SQLResultSetRead[rs]], res += Last[data]];
          res
Out[324]= 0.54
```

If you call `SQLResultSetRead` again it will return `Null` because the end of the result set has been reached.

```
In[325]:= SQLResultSetRead[rs]
```

This closes the result set and the SQL connection.

```
In[326]:= SQLResultSetClose[rs];
          CloseSQLConnection[ conn];
```

SQLResultSet Options

`SQLResultSetOpen` takes an option, `"Mode"`, that controls movement in the result set and whether the result is sensitive to changes in the database.

First, the *DatabaseLink* package is loaded and a connection is made to the *publisher* example database.

```
In[328]:= << DatabaseLink`;
          conn = OpenSQLConnection["publisher"];
```

This opens a result set, but you can only move forwards in this result set.

```
In[330]:= rs = SQLResultSetOpen[
            SQLExecute[conn, "SELECT * FROM roysched", "MaxRows" → 8], Mode → "ForwardOnly"]
Out[330]= SQLResultSet[0, <>, ForwardOnly]
```

setting	*description*
`"ForwardOnly"`	only moving forwards is possible
`"ScrollInsensitive"`	forward and backward moving is possible and result set does not pick up changes to the database
`"ScrollSensitive"`	forward and backward moving is possible and result set picks up changes to the database

Settings of the `Mode` option of `SQLResultSetOpen`.

The `"ForwardOnly"` setting of the `"Mode"` option means that you can only move forwards in the result set and the result set is insensitive to any changes to the database after the result set has been created.

The "ScrollInsensitive" setting of the "Mode" option means that you can move forwards and backwards in the result set and the result set is insensitive to any changes to the database after the result set has been created.

The "ScrollSensitive" setting of the "Mode" option means that you can move forwards and backwards in the result set and the result set is sensitive to any changes to the database after the result set has been created.

You should note that not all databases support moving backwards in the result set or can detect changes in the data.

In addition you use SetOptions to change options of a result set after it has been created. The following sets the direction in which it is expected that result will be retrieved from the result set. This helps the driver to optimize retrieval of data.

```
In[331]:= SetOptions[rs, FetchDirection → "Forward"]
```

setting	description
"FetchDirection"	gives a hint as the direction in which rows will be processed
"FetchSize"	gives a hint as to the number of rows that should be fetched from the database

SQLResultSet options.

Result Set Positions

If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the DatabaseExamples` package, as described in "Using the Example Databases".

SQLResultSetShift[*rs*,*num*]	shift current position by *num* in result set *rs*
SQLResultSetGoto[*rs*,*num*]	move current position to *num* in result set *rs*
SQLResultSetPosition[*rs*]	return current position in result set *rs*
SQLResultSetCurrent[*rs*]	read the row at the current position in result set *rs*

Result set position functions.

A result set is created from a database query, and it can be seen as an array of the rows that match the query. The array actually has two extra positions, one before the first row and one after the last row. When the result set is created, its current position is before the first row.

This loads *DatabaseLink* and creates a result set from a query to the *publisher* database.

```
In[332]:= << DatabaseLink`;
          conn = OpenSQLConnection["publisher"];
          rs = SQLResultSetOpen[SQLExecute[conn, "SELECT * FROM roysched", "MaxRows" → 8]]
Out[334]= SQLResultSet[0, <>, Scrollable]
```

The position is 0, which means that the current position is before the first row.

```
In[335]:= SQLResultSetPosition[rs]
Out[335]= 0
```

If a read is done at the current position, the result is `Null` because there is nothing to read before the first row.

```
In[336]:= SQLResultSetCurrent[rs]
```

The following shifts the result set by two. The result is `True`, which means that there is something to read at the new position.

```
In[337]:= SQLResultSetShift[rs, 2]
Out[337]= True
```

The result set is now positioned at the second row.

```
In[338]:= SQLResultSetPosition[rs]
Out[338]= 2
```

The following reads the row at the current position.

```
In[339]:= SQLResultSetCurrent[rs]
Out[339]= {BS1011, 5001, 50 000, 0.12}
```

By default `SQLResultSetCurrent` maps data types into various *Mathematica* expressions. However, setting the option `"GetAsStrings"` to `True` gets results as string expressions.

```
In[340]:= SQLResultSetCurrent[rs, "GetAsStrings" → True] // InputForm
Out[340]= {"BS1011", "5001", "50000", "0.12"}
```

Now an absolute move is carried out to the eighth row. The result is `True`, which tells you there is something to be read.

```
In[341]:= SQLResultSetGoto[rs, 8]
Out[341]= True
```

This reads the last row in the result set.

```
In[342]:= SQLResultSetCurrent[rs]
Out[342]= {BS1001, 5001, 7000, 0.16}
```

Now a shift of one is done and the result is `False`. This means that there is nothing to be read from this position.

```
In[343]:= SQLResultSetShift[rs, 1]
Out[343]= False
```

The current position is nine, which means that the current position is after the last row.

```
In[344]:= SQLResultSetPosition[rs]
Out[344]= 9
```

If a read is done the result is `Null`; there is nothing to read after the last row.

```
In[345]:= SQLResultSetCurrent[rs]
```

`SQLResultSetShift` can take a negative shift. If the result set allows moving backwards, this will shift backwards. `SQLResultSetGoto` also can take negative settings, these are interpreted as counting from the end of the result set. The following table summarizes how various arguments work.

`SQLResultSetShift[`*rs*`,-`*num*`]`	shift *num* positions to the left in the result set *rs*
`SQLResultSetGoto[`*rs*`,0]`	move to before the first row in the result set *rs*
`SQLResultSetGoto[`*rs*`,3]`	move to the third row in the result set *rs*
`SQLResultSetGoto[`*rs*`,-2]`	move to the second row from the end in the result set *rs*
`SQLResultSetGoto[`*rs*`,-1]`	move to last row in the result set *rs*
`SQLResultSetGoto[`*rs*`,Infinity]`	move to after the last row in the result set *rs*

Examples of result set position functions.

This closes the result set and the SQL connection.

```
In[346]:= SQLResultSetClose[rs];
          CloseSQLConnection[ conn];
```

`SQLResultSetRead[`*rs*`]` can be seen as equivalent to `SQLResultSetShift[`*rs*`, 1]; SQLResultSetCurrent[`*rs*`]`.

Advanced Result Set Operations

This section discusses advanced result set operations.

If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

`SQLResultSetTake[rs, spec]`	use specification *spec* to read from the result set *rs*
`SQLResultSetRead[rs, -num]`	shift current position by *num* in the result set *rs*
`SQLResultSetColumnNames[rs]`	return the names of the columns in the result set *rs*

Advanced result set functions.

This loads *DatabaseLink* and creates a result set from a query to the *publisher* database.

```
In[348]:= << DatabaseLink`;
          conn = OpenSQLConnection["publisher"];
          rs = SQLResultSetOpen[SQLExecute[conn, "SELECT * FROM roysched", "MaxRows" → 8]]
Out[350]= SQLResultSet[0, <>, Scrollable]
```

This shows the rows that are in the result set.

```
In[351]:= SQLExecute[conn, "SELECT * FROM roysched", "MaxRows" → 8] // TableForm
```

BS1011	0	5000	0.1
BS1011	5001	50 000	0.12
CP5018	0	2000	0.1
CP5018	2001	4000	0.12
CP5018	4001	50 000	0.16
BS1001	0	1000	0.1
BS1001	1001	5000	0.12
BS1001	5001	7000	0.16

Out[351]=

The following gets rows two through four.

```
In[352]:= SQLResultSetTake[ rs, {2, 4}]
Out[352]= {{BS1011, 5001, 50 000, 0.12}, {CP5018, 0, 2000, 0.1}, {CP5018, 2001, 4000, 0.12}}
```

After the read, the position is at the fourth row.

```
In[353]:= SQLResultSetPosition[rs]
Out[353]= 4
```

`SQLResultSetTake` can take from the end of the result set. The following reads the last three rows of the result set.

```
In[354]:= SQLResultSetTake[ rs, {-3, -1}]
Out[354]= {{BS1001, 0, 1000, 0.1}, {BS1001, 1001, 5000, 0.12}, {BS1001, 5001, 7000, 0.16}}
```

```
In[355]:= SQLResultSetPosition[rs]
Out[355]= 8
```

`SQLResultSetRead` can also take a negative number. This means that it shifts one position to the left and reads. This is repeated till the requested number has been read. The following goes to the end of the result set and then reads the previous four rows.

```
In[356]:= SQLResultSetGoto[rs, Infinity];
          SQLResultSetRead[ rs, -4]
Out[357]= {{BS1001, 5001, 7000, 0.16}, {BS1001, 1001, 5000, 0.12},
           {BS1001, 0, 1000, 0.1}, {CP5018, 4001, 50 000, 0.16}}
```

After the read, the current position is the last thing that was read.

```
In[358]:= SQLResultSetPosition[rs]
Out[358]= 5
```

By default `SQLResultSetTake` maps data types into various *Mathematica* expressions. However, setting the option `"GetAsStrings"` to `True` gets results as string expressions.

```
In[359]:= SQLResultSetTake[rs, {2, 3}, "GetAsStrings" → True] // InputForm
Out[359]= {{"BS1011", "5001", "50000", "0.12"}, {"CP5018", "0", "2000", "0.1"}}
```

Finally, you can get the names of the columns in a result set by using `SQLResultSetColumnNames`.

```
In[360]:= SQLResultSetColumnNames[rs]
Out[360]= {{ROYSCHED, TITLE_ID}, {ROYSCHED, LORANGE}, {ROYSCHED, HIRANGE}, {ROYSCHED, ROYALTY}}
```

This closes the result set and the SQL connection.

```
In[361]:= SQLResultSetClose[rs];
          CloseSQLConnection[ conn];
```

Result Set Examples

This section discusses common examples of result set operations.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

```
In[363]:= << DatabaseLink`;
          conn = OpenSQLConnection["publisher"];
          rs = SQLResultSetOpen[SQLExecute[conn, "SELECT * FROM roysched", "MaxRows" → 8]]
Out[365]= SQLResultSet[0, <>, Scrollable]
```

This shows the rows that are in the result set.

```
In[366]:= SQLExecute[conn, "SELECT * FROM roysched", "MaxRows" → 8] // TableForm
```

BS1011	0	5000	0.1
BS1011	5001	50 000	0.12
CP5018	0	2000	0.1
CP5018	2001	4000	0.12
CP5018	4001	50 000	0.16
BS1001	0	1000	0.1
BS1001	1001	5000	0.12
BS1001	5001	7000	0.16

One common operation is to iterate over all the rows, operating on each of the rows one at a time. The following example sums the last element of each row.

```
In[367]:= res = 0; While[ListQ[ data = SQLResultSetRead[rs]], res += data[[-1]]];
          res
Out[368]= 0.98
```

The following resets the result set to the beginning.

```
In[369]:= SQLResultSetGoto[rs, 0]
Out[369]= False
```

This example extracts every second row of the result set. It does this by shifting and reading the current row. The result is formed by using `Reap` and `Sow`.

```
In[370]:= Last[Reap[ While[SQLResultSetShift[rs, 2], Sow[SQLResultSetCurrent[rs]]]]]
Out[370]= {{{BS1011, 5001, 50 000, 0.12}, {CP5018, 2001, 4000, 0.12},
            {BS1001, 0, 1000, 0.1}, {BS1001, 5001, 7000, 0.16}}}
```

This closes the result set and the SQL connection.

```
In[371]:= SQLResultSetClose[rs];
          CloseSQLConnection[ conn];
```

Performance

Batch Operation

When large amounts of data are being transferred between *Mathematica* and a database, you may find that the operations are slow. In this case it may be advantageous to use a batch operation mode. If many small operations are being repeated, this will be likely to improve the performance. This section will demonstrate how to use batch statements.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

This loads *DatabaseLink* and connects to the *demo* database.

```
In[373]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

This creates a simple table. The table name is *BATCH* with columns *X* and *Y*. The data types for *X* and *Y* are integers.

```
In[375]:= table = SQLTable["BATCH"];
          cols = {SQLColumn["X", "DataTypeName" -> "Integer"],
             SQLColumn["Y", "DataTypeName" -> "Integer"]};
          SQLCreateTable[conn, table, cols];
```

This generates data to insert into the table. *X* will range from 1 to 10,000 and *Y* will range from 1 to $10,000^2$. The data consists of 10,000 rows.

```
In[378]:= data1 = {table, SQLArgument @@ cols, SQLArgument[#, #^2]} & /@ Range[10 000];
```

This uses `Map` to execute the SQL insert 10,000 times.

```
In[379]:= AbsoluteTiming[
           SQLExecute[conn, "INSERT INTO `1` (`2`) VALUES (`3`)", #] & /@ data1 ;]
Out[379]= {25.6562133 Second, Null}
```

This demonstrates that 10,000 elements have been inserted.

```
In[380]:= Length[SQLSelect[conn, "BATCH"]]
Out[380]= 10 000
```

This uses a batch mode to insert the data. This is done by passing a list of arguments to `SQLExecute`. Each element of the list contains an `SQLTable` expression, an `SQLArgument` expression with the sequence of column names, and an `SQLArgument` expression with the pairs of values.

```
In[381]:= AbsoluteTiming[SQLExecute[conn, "INSERT INTO `1` (`2`) VALUES (`3`)", data1];]
Out[381]= {7.0755258 Second, Null}
```

```
In[382]:= Length[SQLSelect[conn, "BATCH"]]
Out[382]= 20 000
```

The batch operation has reduced the time by more than a factor of three. This is because it has done the insert operation in one call rather than 10,000 smaller calls.

The new table is dropped and the connection closed.

```
In[383]:= SQLDropTable[conn, "BATCH"];
          CloseSQLConnection[conn];
```

Simplifying Substitution Patterns

Simplifying substitution patterns is another technique for increasing performance. This will be demonstrated using a table identical to the previous example.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

This loads *DatabaseLink* and connects to the *demo* database.

```
In[385]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

This creates a simple table. The table name is *BATCH* with column *X* and *Y*. The data types for *X* and *Y* are integers.

```
In[387]:= table = SQLTable["BATCH"];
          cols = {SQLColumn["X", "DataTypeName" -> "Integer"],
             SQLColumn["Y", "DataTypeName" -> "Integer"]};
          SQLCreateTable[conn, table, cols];
```

Since the table and columns are always the same for this call, it is faster to place them directly into a prepared statement rather than substitute values for them each time. It is also faster to

use a list for the values rather than an SQLArgument expression. This creates the data to be used for the test.

```
In[390]:= data2 = {#, #^2} & /@ Range[10 000];
```

Now the insert operation is carried out. This has reduced the time for the operation by a factor of more than 14.

```
In[391]:= AbsoluteTiming[
           SQLExecute[conn, "INSERT INTO BATCH (X,Y) VALUES (`1`,`2`)", data2];]
Out[391]= {1.7381556 Second, Null}
```

This confirms that 10,000 inserts have been carried out.

```
In[392]:= Length[SQLSelect[conn, "BATCH"]]
Out[392]= 10 000
```

A final performance improvement can be done by using JDBC syntax for substitutions. This limits dynamic values of the types of data that may be substituted to only `Real`, `Integer`, `String`, `True`, `False`, `Null`, `SQLBinary`, and `SQLDateTime`. It also uses '?' instead of the `` `1` `` notation (the first parameter in the list will replace the first question mark and so on).

Now the operation runs nearly 50 times faster than the original simple operation of repeated inserts.

```
In[393]:= AbsoluteTiming[SQLExecute[conn, "INSERT INTO BATCH (X,Y) VALUES (?,?)", data2];]
Out[393]= {0.5451894 Second, Null}
```

```
In[394]:= Length[SQLSelect[conn, "BATCH"]]
Out[394]= 20 000
```

The *Mathematica* command `SQLInsert` uses this last technique. When you pass a table of values as a parameter to `SQLInsert`, it uses the fastest way to insert the data.

```
In[395]:= AbsoluteTiming[SQLInsert[conn, "BATCH", {"X", "Y"}, data2];]
Out[395]= {0.5754777 Second, Null}
```

```
In[396]:= Length[SQLSelect[conn, "BATCH"]]
Out[396]= 30 000
```

This drops the table and closes the connection.

```
In[397]:= SQLDropTable[conn, "BATCH"];
          CloseSQLConnection[conn];
```

Result Sets

When many rows of data are returned from a database query, they may require a significant amount of memory to hold. For your purposes, you may not need to hold all of the data. You may need to use each row individually as part of a computation or you may only need to sample the rows. In cases such as these, you may find the result set functionality beneficial. This is described in "Result Sets".

Descriptive Commands

If the database is very large, then certain descriptive commands, such as querying the number of tables with `SQLTables`, can be slow. In this case, if some of the tables in the database have been placed into catalogs, performance can be improved by using the `"Catalog"` or `"Schema"` options. These are described in "Table Structure: Table Description "and" Column Structure: Column Description".

Connection Pools

Database connection pools are a common way to improve the performance of database operations. They can be useful because creating a new connection can easily take several seconds to establish; this is a problem when the database operation is one that only needs a few milliseconds. *DatabaseLink* provides a connection pool mechanism built on top of the Apache Commons DBCP, http://jakarta.apache.org/commons/dbcp/index.html.

Working with Connection Pools

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

To create a connection from a pool you can set the `UseConnectionPool` option of `OpenSQLConnection`. Here is an example.

First, the *DatabaseLink* package is loaded. Then a connection using a pool is made to the *publisher* example database.

```
In[399]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "publisher", UseConnectionPool → True]
Out[399]= SQLConnection[publisher, 2, Open]
```

Instead of using the `UseConnectionPool` option, you could set the default value `$SQLUseConnectionPool` to `True`. When *DatabaseLink* loads it is `False`.

```
In[400]:= $SQLUseConnectionPool
Out[400]= False
```

`OpenSQLConnection[`*src*`,UseConnectionPool->True]`	connect to a data source using a connection pool
`$SQLUseConnectionPool`	whether to always use connection pools
`SQLConnectionPools[]`	information on all active connection pools
`SQLConnectionPools[`*conn*`]`	information on pool for connection *conn*
`SetSQLConnectionPoolOptions[`*pool*`]`	set options for connection pool *pool*
`SQLConnectionPoolClose[`*pool*`]`	close the connection pool *pool*

Commands for working with connection pools.

This shows all the connection pools that have been created; there is only one.

```
In[401]:= SQLConnectionPools[]
Out[401]= {SQLConnectionPool[« JavaObject[org.apache.commons.dbcp.BasicDataSource] »,
            JDBC[HSQL(Standalone), C:\Documents and Settings\twj.WRI\Application
               Data\Mathematica\DatabaseResources\Examples\publisher], 1,
            Catalog → Automatic, Description → Connection to HSQL publisher database for demos.,
            Location → C:\Program Files\Wolfram
               Research\Mathematica\6.0_Thin\SystemFiles\Links\DatabaseLink\DatabaseResources\
               publisher.m, Name → publisher, Password → None, Properties → {},
            ReadOnly → Automatic, RelativePath → False, TransactionIsolationLevel → Automatic,
            UseConnectionPool → True, Username → None, Version → 2.]}
```

This shows the connection pool used to connect to the publisher database. You can see some of the options that the connection pool is using.

```
In[402]:= pool = SQLConnectionPools[conn]

Out[402]= SQLConnectionPool[« JavaObject[org.apache.commons.dbcp.BasicDataSource] »,
  JDBC[HSQL(Standalone), C:\Documents and Settings\twj.WRI\Application
     Data\Mathematica\DatabaseResources\Examples\publisher], 1,
  Catalog → Automatic, Description → Connection to HSQL publisher database for demos.,
  Location → C:\Program Files\Wolfram
     Research\Mathematica\6.0_Thin\SystemFiles\Links\DatabaseLink\DatabaseResources\publisher
     .m, Name → publisher, Password → None, Properties → {}, ReadOnly → Automatic,
  RelativePath → False, TransactionIsolationLevel → Automatic,
  UseConnectionPool → True, Username → None, Version → 2.]
```

This closes the connection pool, and it also closes any connections that the pool is using.

```
In[403]:= SQLConnectionPoolClose[pool]
```

Connection Pool Options

There are a number of options that control how the connection pool operates. This example shows how to work with them.

First, the *DatabaseLink* package is loaded. Then a connection using a pool is made to the *publisher* example database.

```
In[404]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "publisher", UseConnectionPool → True]

Out[405]= SQLConnection[publisher, 1, Open]
```

This shows all the connection pools that have been created; there is only one.

```
In[406]:= SQLConnectionPools[]

Out[406]= {SQLConnectionPool[« JavaObject[org.apache.commons.dbcp.BasicDataSource] »,
   JDBC[HSQL(Standalone), C:\Documents and Settings\User\Application
      Data\Mathematica\DatabaseResources\Examples\publisher], 1,
   Catalog → Automatic, Description → Connection to HSQL publisher database for demos.,
   Location → C:\Program Files\Wolfram
      Research\Mathematica\6.0\SystemFiles\Links\DatabaseLink\DatabaseResources\publisher.m,
   Name → publisher, Password → None, Properties → {}, ReadOnly → Automatic,
   RelativePath → False,
   TransactionIsolationLevel → Automatic,
   UseConnectionPool → True, Username → None, Version → 2.]}
```

This shows the connection pool used to connect to the publisher database. You can see some of the options that the connection pool is using.

```
In[407]:= SQLConnectionPools[conn]

Out[407]= SQLConnectionPool[« JavaObject[org.apache.commons.dbcp.BasicDataSource] »,
  JDBC[HSQL(Standalone), C:\Documents and Settings\User\Application
     Data\Mathematica\DatabaseResources\Examples\publisher], 1,
  Catalog → Automatic, Description → Connection to HSQL publisher database for demos.,
  Location → C:\Program Files\Wolfram
     Research\Mathematica\6.0\SystemFiles\Links\DatabaseLink\DatabaseResources\publisher.m,
  Name → publisher, Password → None, Properties → {}, ReadOnly → Automatic,
  RelativePath → False,
  TransactionIsolationLevel → Automatic,
  UseConnectionPool → True, Username → None, Version → 2.]
```

This sets the `MaxActiveConnections` option of this connection pool.

```
In[408]:= SetSQLConnectionPoolOptions[SQLConnectionPools[conn], MaxActiveConnections → 8]

Out[408]= SQLConnectionPool[« JavaObject[org.apache.commons.dbcp.BasicDataSource] »,
  JDBC[HSQL(Standalone), C:\Documents and Settings\User\Application
     Data\Mathematica\DatabaseResources\Examples\publisher], 1,
  Catalog → Automatic, Description → Connection to HSQL publisher database for demos.,
  Location → C:\Program Files\Wolfram
     Research\Mathematica\6.0\SystemFiles\Links\DatabaseLink\DatabaseResources\publisher.m,
  Name → publisher, Password → None, Properties → {}, ReadOnly → Automatic,
  RelativePath → False,
  TransactionIsolationLevel → Automatic,
  UseConnectionPool → True, Username → None, Version → 2.]
```

```
In[409]:= CloseSQLConnection[conn]
```

`SQLConnectionPools[]`	information on all active connection pools
`SQLConnectionPools[`*conn*`]`	information on pool for connection *conn*
`SetSQLConnectionPoolOptions[`*pool*`]`	set options for connection pool *pool*
`CloseConnectionPool[`*pool*`]`	close the connection pool *pool*

Functions for working with connection pool options.

option name	
`"MaxActiveConnections"`	maximum number of connections to keep in the pool
`"MinIdleConnections"`	minimum number of idle connections to keep in the pool
`"MaxIdleConnections"`	maximum number of idle connections to keep in the pool
`"Catalog"`	location of the database catalog
`"ReadOnly"`	set the connection to be read only
`"TransactionIsolationLevel"`	set transaction isolation for the connection

Connection pool options.

Transactions

Some database operations involve carrying out a sequence of database commands. For example, information in two different tables may need to be updated. In these cases it may be very important that if one update is carried out, the other is also. If only one is done, it may leave the data inconsistent. You can use database transactions to ensure that all the operations are carried out. In addition, you can use transactions as a way of backing out of the middle of a sequence of operations. This tutorial will demonstrate how to use transactions.

> If you find that the examples in this tutorial do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

`SQLBeginTransaction[`*conn*`]`	begin an SQL transaction
`SQLCommitTransaction[`*conn*`]`	permanently commit an SQL transaction to the database
`SQLRollbackTransaction[`*conn*`]`	terminate an SQL transaction (do not change the database)

Functions for executing SQL transactions.

This loads *DatabaseLink* and connects to the *demo* database.

```
In[410]:= Needs["DatabaseLink`"];
          conn = OpenSQLConnection[ "demo"];
```

This creates a table to use for testing.

```
In[412]:= table = SQLTable["TEST"];
          cols = {SQLColumn["X", "DataTypeName" -> "Integer"],
             SQLColumn["Y", "DataTypeName" -> "Integer"]};
          SQLCreateTable[conn, table, cols];
          SQLInsert[conn, "TEST", {"X", "Y"}, {5, 6}];
```

This uses `SQLSelect` to view data in the *TEST* table. There is one row.

```
In[416]:= SQLSelect[conn, "TEST", "ShowColumnHeadings" → True] // TableForm

Out[416]= X    Y
          5    6
```

`SQLBeginTransaction` is used to start a transaction.

```
In[417]:= SQLBeginTransaction[conn]
```

Next, two different insert operations are carried out.

```
In[418]:= SQLInsert[conn, "TEST", {"X", "Y"}, {61, 80}];
          SQLInsert[conn, "TEST", {"X", "Y"}, {72, 5}];
```

This shows that two rows have been inserted.

```
In[420]:= SQLSelect[conn, "TEST"] // TableForm

          5    6
Out[420]= 61   80
          72   5
```

If SQLRollbackTransaction is used, the database is returned to the point before the transaction began. The two rows are no longer present.

```
In[421]:= SQLRollbackTransaction[conn];
          SQLSelect[conn, "TEST"] // TableForm

Out[422]= 5    6
```

A transaction is closed when it is rolled back. If any more transactions are required, a new transaction must be started. Here, a new transaction is started and the two rows are reinserted.

```
In[423]:= SQLBeginTransaction[conn];
          SQLInsert[conn, "TEST", {"X", "Y"}, {111, 141}];
          SQLInsert[conn, "TEST", {"X", "Y"}, {190, 1}];
```

This uses SQLCommitTransaction to commit the data permanently.

```
In[426]:= SQLCommitTransaction[conn];
          SQLSelect[conn, "TEST"] // TableForm

          5     6
Out[427]= 111   141
          190   1
```

A transaction is closed when it is committed. If any more transactions are required, a new transaction must be started. In addition, once a transaction has been committed, it cannot be rolled back. Transactions may be split up using an SQLSavepoint; a rollback can be made to a specific savepoint.

The following begins a transaction and inserts some data.

```
In[428]:= SQLBeginTransaction[conn];
          SQLInsert[conn, "TEST", {"X", "Y"}, {22, 11}];
```

A savepoint is created.

```
In[430]:= savepoint = SQLSetSavepoint[conn, "savepoint1"]

Out[430]= SQLSavepoint[«JavaObject[org.hsqldb.jdbc.jdbcSavepoint]»]
```

Here some more data is inserted into the database.

```
In[431]:= SQLInsert[conn, "TEST", {"X", "Y"}, {17, 22}];
          SQLSelect[conn, "TEST"] // TableForm
```

```
           5     6
           111   141
Out[432]=  190   1
           22    11
           17    22
```

The transaction is rolled back to the savepoint using `SQLRollbackTransaction`.

```
In[433]:= SQLRollbackTransaction[conn, savepoint]
```

This shows that the last insert has not taken place.

```
In[434]:= SQLSelect[conn, "TEST"] // TableForm
```

```
           5     6
           111   141
Out[434]=  190   1
           22    11
```

This drops the *TEST* table and closes the connection.

```
In[435]:= SQLDropTable[conn, "TEST"];
          CloseSQLConnection[conn];
```

Transaction Isolation

When working with database transactions with more than one concurrent user various problems with reading data can occur. These problems are can be termed as 'dirty reads', 'non-repeatable reads', and 'phantom reads'. There are two types of solution to these problems, one involves setting options for the database connection to isolate transactions, and the other involves other checks on data or instance by checking timestamps. Each of these strategies have advantages and disadvantages, for example, setting database options can degrade the performance of the database for concurrent usage.

The actual details of these strategies are really outside the scope of this documentation. However, *DatabaseLink* has a number of ways to set options of the connection to help isolate transac tions. This is done with the `TransactionIsolationLevel` option of `OpenSQLConnection`. This option can also be set for an existing connection with `SetOptions`.

setting	*description*
`ReadUncommitted`	no isolation
`ReadCommitted`	prevent dirty reads
`RepeatableRead`	prevent dirty reads and non repeatable reads
`Serializable`	prevent dirty reads, non repeatable reads, and phantom reads

Settings of the `TransactionIsolationLevel` option.

Secure Socket Layer (SSL)

Secure Socket Layer (SSL) is a protocol for providing secure transactions between servers and clients. It uses a certificate to identify one or both ends of the transaction. It can be useful for database communications to protect any authentication information, such as usernames and passwords, as well as the actual data itself.

Some databases support SSL and some do not. To know if your database supports SSL, you need to study the documentation for your database and work with the administrator of the database. If your database can be configured to use SSL with JDBC, it should be possible to configure *DatabaseLink* to communicate with the database using SSL.

One database that does support SSL is MySQL, and it is possible for *DatabaseLink* to communicate with a MySQL database using SSL. You will need to configure the database to provide SSL communications and generate a certificate. To do this you will need to work with the administrator of your database.

There are typically four stages to setting up SSL to work with a MySQL database.

1. Get a certificate of authority.
2. Generate a truststore file.
3. Configure Java to use the truststore.
4. Configure the connection to use SSL.

The administrator of the server should be able to provide the certificate of authority, suppose this is called `CA.cer`.

You need to generate the truststore file. This can be done with the `keytool` executable that is part of a Java Runtime Environment (JRE). You can use the version included in the JRE that ships with *Mathematica*. To generate the truststore file, you would need to execute the following in some type of shell (e.g. a command prompt on Windows).

```
keytool -import -file CA.cer -keystore truststore
```

This will generate the file `truststore`.

The next stage is to modify your Java command line for *J/Link* to refer to the `truststore` file. This can be done by adding the following settings, in which you need to give the full pathname to the `truststore` file that was generated.

```
-Djavax.net.ssl.trustStore=c:\java-examples\truststore
-Djavax.net.ssl.trustStorePassword=keystore
```

If you are running *Mathematica* inside a web server, such as web*Mathematica*, you will need to add these settings to the server that launches Java by following your server documentation. If you are running *Mathematica* in a stand-alone fashion, you can add the settings to the options of Java by executing the following before you load *DatabaseLink*.

```
Needs[ "JLink`"];
SetOptions[InstallJava, JVMArguments ->"-Djavax.net.ssl.trustStore=c:\java-
examples\truststore -Djavax.net.ssl.trustStorePassword=keystore"]
```

Finally, you need to modify the URL that connects to the database. This can be done by placing an extra parameter with a '?', as shown in the following.

```
OpenSQLConnection[ JDBC[ "com.mysql.jdbc.Driver",
"databases:1234/conn_test?useSSL=true"], "Username" -> "test"]
```

It should be noted that not all databases support SSL and that databases other than MySQL that do support SSL may need to be configured in a different way to work with *DatabaseLink*.

Examples

Command Cache

This example shows how to use a private database to store *Mathematica* commands and query the data from Java and *Mathematica*.

> If you find that the examples in this section do not work as shown, you may need to install or restore the example database with the `DatabaseExamples`` package, as described in "Using the Example Databases".

The example code is loaded from the `Examples` subdirectory of *DatabaseLink*.

```
In[437]:= << DatabaseLink`Examples`CommandCache`
```

The command cache allows you to store *Mathematica* expressions as typeset box expressions data in a database. `StoreCommand` is used to store the boxes in the database.

```
In[438]:= StoreCommand[MakeBoxes[2 + 2]]
Out[438]= RowBox[{2, +, 2}]
```

The data can then be retrieved from the database using `CommandCache[]`.

```
In[439]:= CommandCache[]
Out[439]= {{0, SQLExpr[RowBox[{2, +, 2}]], Plus[2, 2],
  SQLBinary[{71, 73, 70, 56, 57, 97, 31, 0, 17, 0, 240, 0, 0, 0, 0, 0, 255, 255, 255, 33, 249, 4,
    1, 0, 0, 1, 0, 44, 0, 0, 0, 0, 31, 0, 17, 0, 0, 2, 46, 140, 143, 169, 203, 237, 15, 163,
    156, 180, 42, 128, 41, 6, 140, 27, 15, 129, 96, 103, 141, 215, 98, 38, 41, 178, 6, 173,
    235, 172, 91, 166, 198, 71, 42, 223, 232, 124, 94, 188, 5, 12, 10, 135, 68, 68, 1, 0, 59}],
  SQLDateTime[2004, 8, 4, 16, 57, 56.7309]}}
```

Several attributes of each command are stored in the database. Each command is given an ID, generated when it is stored. The expression is stored as a string formatted with `FullForm`; this allows it to be reused in *Mathematica*. In addition, an image of the expression is saved as well as the time at which the data was stored. `GetCommandAttributes` can be used to get the attribute names. These can be used to filter the data returned.

```
In[440]:= GetCommandAttributes[]
Out[440]= {ID, EXPR, FULLFORM, IMAGE, USED, *}
```

```
In[441]:= CommandCache[{"ID", "FULLFORM"}]
Out[441]= {{0, Plus[2, 2]}}
```

The following example finds all results that contain `Power`.

```
In[442]:= StoreCommand[MakeBoxes[2^2]]
Out[442]= SuperscriptBox[2, 2]
```

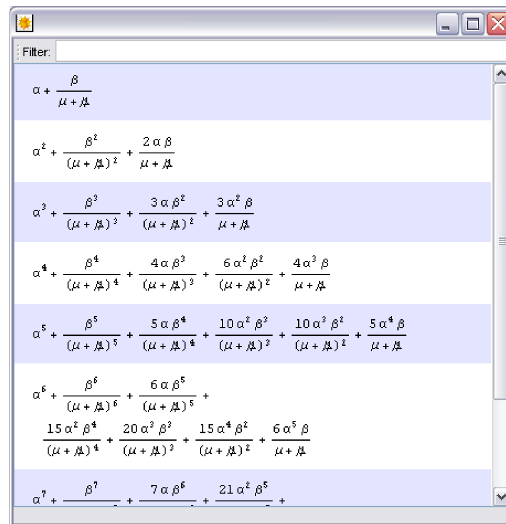
```
In[443]:= CommandCache[{"ID", "FULLFORM"}]
Out[443]= {{0, Plus[2, 2]}, {1, Power[2, 2]}}
```

```
In[444]:= CommandCache["Power", {"ID", "FULLFORM"}]
Out[444]= {{1, Power[2, 2]}}
```

A command can also be retrieved using its ID.

```
In[445]:= CommandCache[1]
Out[445]= {{1, SQLExpr[SuperscriptBox[2, 2]], Power[2, 2],
   SQLBinary[{71, 73, 70, 56, 57, 97, 17, 0, 17, 0, 240, 0, 0, 0, 0, 0, 255, 255, 255, 33,
     249, 4, 1, 0, 0, 1, 0, 44, 0, 0, 0, 0, 17, 0, 17, 0, 0, 2, 30, 140, 143, 169, 11, 235,
     221, 156, 138, 50, 209, 9, 232, 181, 39, 230, 61, 85, 156, 216, 145, 165, 25, 104,
     228, 151, 161, 238, 11, 187, 5, 0, 59}], SQLDateTime[2004, 8, 4, 16, 58, 8.51198]}}
```

Another feature of this package is a Java GUI you can use to browse the data. It provides functionality for managing the data and pasting the data into a notebook.




```
In[446]:= CommandBrowser[]
Out[446]= - GUIObject -
```

The GUI is automatically updated when new commands are added to the database.

```
In[447]:= Do[StoreCommand[ToBoxes[Expand[(α + β/([illegible] + μ))^i]]], {i, 0, 10}]
```

Likewise, the GUI is automatically updated when commands are removed. `ClearCommandCache` can be used to remove all the data in the command cache.

```
In[448]:= ClearCommandCache[]
Out[448]= 0
```

Graph Database

This example shows a database that stores material generated by *Mathematica.* Here the material involves graphs and a number of properties of these graphs. Even though the number of graphs is not extremely large, generating the properties of these graphs can take a significant amount of time. This demonstrates the value of a database for persistent storage of the results of computations.

Using the Graph Database

To use the graph database you need to load the package.

```
In[449]:= << DatabaseLink`Examples`Graphs`
```

The first time the package is used you will need to run the `RestoreGraphDatabase` command. If you find that the examples in this section do not work as shown, you should also run this command. This command can take a long time to run, but is only necessary once.

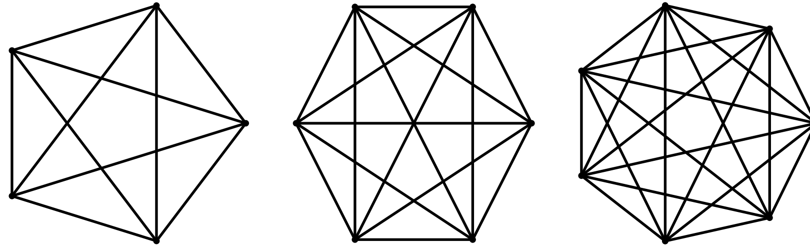
```
In[450]:= RestoreGraphDatabase[]
```

The properties of the graphs stored in the database are given by the `GraphProperties` function.

```
In[451]:= GraphProperties[]
Out[451]= {*, GRAPH, ORDER, EDGES, VERTEXCONNECTIVITY, EDGECONNECTIVITY, DIAMETER, GIRTH,
  NUMBEROFSPANNINGTREES, SPECTRUMLENGTH, SIMPLEQ, CONNECTEDQ, BIPARTITEQ, PLANARQ, REGULARQ,
  EULERIANQ, HAMILTONIANQ, TREEQ, BICONNECTEDQ, COMPLETEQ, PERFECTQ, SELFCOMPLEMENTARYQ}
```

Now, you can make a query from the database. This is done with the `GraphQuery` command. The following returns all complete graphs.

```
In[452]:= GraphQuery[{"COMPLETEQ" → True}]
```

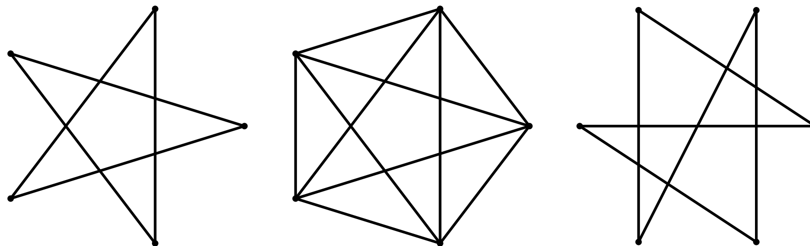


```
Out[452]= {{-Graph:<10, 5, Undirected>-}, {-Graph:<15, 6, Undirected>-}, {-Graph:<21, 7, Undirected>-}}
```

The format used for the graphs is that provided by the *Combinatorica* package. This draws a picture of the graph and also returns a symbolic object that could be used for further computation by *Mathematica*.

The following returns the first three regular graphs.

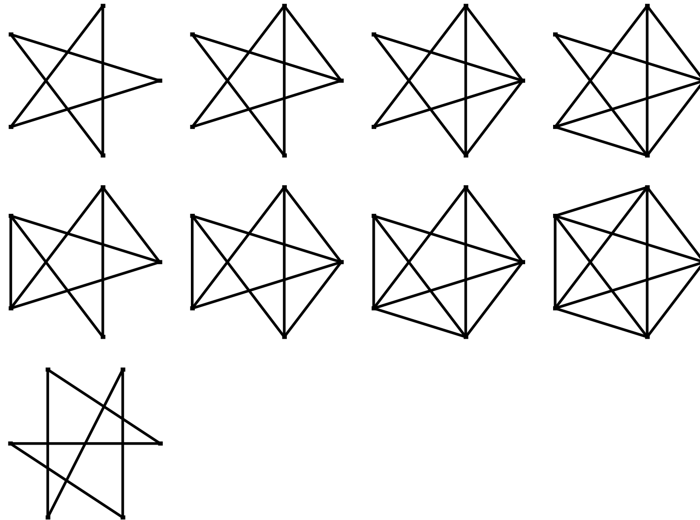
```
In[453]:= GraphQuery[{"REGULARQ" → True}, "MaxHits" → 3]
```



```
Out[453]= {{-Graph:<5, 5, Undirected>-}, {-Graph:<10, 5, Undirected>-}, {-Graph:<6, 6, Undirected>-}}
```

The following finds Hamiltonian graphs, returning their diameter, girth, and edge information. As before, a picture of the graph is also drawn.

```
In[454]:= GraphQuery[{"HAMILTONIANQ" → True}, {"diameter", "girth", "edges"}]
```



```
Out[454]= {{2, 5, 5}, {2, 3, 6}, {2, 3, 7}, {2, 3, 8}, {2, 3, 7}, {2, 3, 8}, {2, 3, 9}, {1, 3, 10}, {3, 6, 6}}
```

One important aspect of this example package is that it shows commands that are specific to the issue of finding graphs rather than general database commands. The details of the database interactions are all placed in the implementation.

The Graph Database Package

The graph database package provides three functions.

`GraphProperties[]`	list all graph properties within the database
`GraphQuery[{`*props*`}]`	search the database for graphs that match *props*
`RestoreGraphDatabase[]`	restore the data in the database (can be slow)

Graph database package functions.

The `RestoreGraphDatabase` function recomputes all the data in the database and can take some time to complete. Generally you do not want to run this, unless you have corrupted the database in some way.

Appendix

Database Reference

HSQLDB

HSQLDB is a relational database engine written in Java that is bundled with *DatabaseLink*, which also contains a JDBC driver and necessary configuration. It offers a small (about 100k), fast database engine, which can run in a variety of ways, including server, in-process, and in-memory modes. *DatabaseLink* is configured to use an in-process stand-alone mode. This makes it very simple to run and use (no special configuration is required). However, it means that nothing else can connect to the database and only one connection to a particular database can be made at any one time (even by multiple copies of *Mathematica*).

To create a new database with HSQLDB, you just need to make a connection to a database that does not already exist, and HSQLDB will create it for you. You could use the Connection Tool, which will deploy a wizard and write a named connection. This is described in "The Database Explorer: The Connection Tool". You could also write a connection file and place this in a `DatabaseResources` directory, as described in "Database Resources: Connection Configuration". Finally, you can use `OpenSQLConnection` as follows. All of these issues are described in "Database Connections".

The following is a sample command that will create a new database called *example*.

```
In[455]:= conn = OpenSQLConnection[JDBC["HSQL(Standalone)",
             ToFileName[{$UserAddOnsDirectory, "Applications", "DatabaseLink", "Examples"},
              "example"]], "Name" → "manualA", "Username" → "sa"]
```

The details of the HSQLDB driver in *DatabaseLink* can be seen as follows.

```
In[456]:= Needs["DatabaseLink`"];
          JDBCDrivers["HSQL(Standalone)"]

Out[457]= JDBCDriver[Name → HSQL(Standalone),
           Driver → org.hsqldb.jdbcDriver, Protocol → jdbc:hsqldb:file:, Version → 1.1,
           Description → HSQL Database Engine (In-Process Mode) - Version 1.8.0.0 -  This mode
              runs the database engine as part of your application program in the
              same Java Virtual Machine. The main drawback is that it is not possible
              by default to connect to the database from outside your application.,
           Location → C:\Documents and Settings\All Users\Application
              Data\Mathematica\Applications\DatabaseLink\DatabaseResources\hsqldbstandalone.m]
```

To connect to an HSQLDB database you would typically give the filename, username, and password as in the following example.

```
In[458]:= OpenSQLConnection[JDBC["HSQL(Standalone)", "file"], "Username" -> "user",
  "Password" -> "password"]
```

For more information, see hsqldb.sourceforge.net/.

MySQL

The MySQL database server is an extremely popular open source database. It is used in many different types of applications. *DatabaseLink* comes configured with a driver for MySQL.

If you want to create a new database for MySQL, you should contact the server administrator.

The details of the MySQL driver in *DatabaseLink* can be seen as follows.

```
In[459]:= Needs["DatabaseLink`"];
          JDBCDrivers["MySQL(Connector/J)"]

Out[460]= JDBCDriver[Name → MySQL(Connector/J), Driver → com.mysql.jdbc.Driver,
  Protocol → jdbc:mysql://, Version → 1.1, Description →
   MySQL using Connector/J - Version 3.1.10 - This supports all known MySQL server versions.,
  Location → C:\Documents and Settings\All Users\Application
     Data\Mathematica\Applications\DatabaseLink\DatabaseResources\mysql.m]
```

To connect to a MySQL database you would typically set the server, database, username, and password as in the following example.

```
In[461]:= OpenSQLConnection[
  JDBC["MySQL(Connector/J)", "server/database"], "Username" -> "user",
  "Password" -> "password"]
```

For more information, see www.mysql.com/.

ODBC

Open Database Connectivity (ODBC) is a general way to connect to SQL databases that is supported in a number of operating systems, particularly Microsoft Windows. *DatabaseLink* comes configured with a driver for ODBC connections.

Under Windows, there is an ODBC Data Source Administrator that can be used to connect to a variety of different databases. Database Connections: ODBC Connections shows how to connect to a database using ODBC.

The details of the ODBC driver in *DatabaseLink* can be seen as follows.

```
In[462]:= Needs["DatabaseLink`"];
          JDBCDrivers["ODBC(DSN)"]

Out[463]= JDBCDriver[Name → ODBC(DSN), Driver → sun.jdbc.odbc.JdbcOdbcDriver,
           Protocol → jdbc:odbc:, Version → 1.1, Description →
            JDBC-ODBC Bridge distributed with the Sun JVM.  This driver only works on Windows.,
           Location → C:\Documents and Settings\All Users\Application
              Data\Mathematica\Applications\DatabaseLink\DatabaseResources\odbcdsn.m]
```

To connect to an ODBC database you would typically use a data source name as in the following example.

```
In[464]:= OpenSQLConnection[ JDBC["ODBC(DSN)", "datasource"]]
```

SQL Server

Support for Microsoft SQL Server is provided by the jTDS driver.

The details of the SQL Server driver in *DatabaseLink* can be seen as follows.

```
In[465]:= Needs["DatabaseLink`"];
          JDBCDrivers["Microsoft SQL Server (jTDS)"]
```

To connect to a Microsoft SQL Server database you would typically set the server, database, username, and password as in the following example.

```
In[467]:= OpenSQLConnection[ JDBC["Microsoft SQL Server (jTDS)", "server/database"],
           "Username" -> "user", "Password" -> "password"]
```

For more information, see: jtds.sourceforge.net/ an open source driver for Microsoft SQL Server and Sybase.

Sybase

Support for Sybase is provided by the jTDS driver.

The details of the Sybase driver in *DatabaseLink* can be seen as follows.

```
In[468]:= Needs["DatabaseLink`"];
          JDBCDrivers["Sybase (jTDS)"]
```

To connect to a Microsoft SQL Server database you would typically set the server, database, username, and password as in the following example.

```
In[470]:= OpenSQLConnection[JDBC["Sybase (jTDS)", "server/database"],
            "Username" -> "user", "Password" -> "password"]
```

For more information, see: jtds.sourceforge.net/ an open source driver for Microsoft SQL Server and Sybase.

Other Databases

DatabaseLink can connect to any other type of database with a JDBC driver. You can install the driver by following the instructions in Database Connections: JDBC Connections and Database Resources: JDBC Configuration.

Information on how to obtain and install drivers as well as configuring connection information for a number of databases is available at www.wolfram.com/solutions/connections/database/vendors.html.

JDBC

The Java Database Connectivity API: java.sun.com/products/jdbc/.

Using the Example Databases

DatabaseLink contains a number of example databases (many use HSQLDB). These allow you to try examples in the documentation and learn the details of working with databases in *Mathematica*. The examples are configured to run in `$UserBaseDirectory/DatabaseResources/Examples` (they cannot reside inside the main *Mathematica* installation directory). To run these examples you will need to install them. You can do this by copying the files or by running the command `DatabaseExamplesBuild` from the ``DatabaseLink`DatabaseExamples` `` package. This function will install the examples (if necessary) or restore them to their original state.

The following shows the location of the database examples on this computer.

```
In[471]:= ToFileName[{$UserBaseDirectory, "DatabaseResources"}, "Examples"]

Out[471]= C:\Documents and Settings\twj.WRI\Application Data\Mathematica\DatabaseResources\Examples
```

DatabaseLink`DatabaseExamples`	load the DatabaseExamples` package
DatabaseExamplesBuild[" "]	install and restore the database examples

Using the DatabaseExamples` package.

You must run DatabaseExamplesBuild the first time you want to use the documentation, and after you have been working with the example databases and want to restore them to their original state.

First, the package is loaded.

```
In[472]:= << DatabaseLink`DatabaseExamples`;
```

Then the examples are installed, if necessary, or restored to their original state.

```
In[473]:= DatabaseExamplesBuild[]
```

If you want to install the examples by hand, copy the Examples directory from inside the *DatabaseLink* installation directory (typically this is $InstallationDirectory/SystemFiles/Links/DatabaseLink) to $UserBaseDirectory/DatabaseResources.