SYBASE iAnywhere

SQL Anywhere® Server Programming

June 2008

Version 11.0.0

Copyright and trademarks

Contents

About this book

Subject

This book describes how to build and deploy database applications using the C, C++, Java, PHP, Perl, Python, and .NET programming languages such as Visual Basic and Visual C#. A variety of programming interfaces such as ADO.NET and ODBC are described.

Audience

This book is intended for application developers writing programs that work directly with one of the SQL Anywhere interfaces.

About the SQL Anywhere documentation

The complete SQL Anywhere documentation is available in three formats that contain identical information.

- **HTML Help** The online Help contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools. The online Help is updated with each release of the product.

  If you are using a Microsoft Windows operating system, the online Help is provided in HTML Help (CHM) format. To access the documentation, choose **Start** » **Programs** » **SQL Anywhere 11** » **Documentation** » **Online Books**.

  The administration tools use the same online documentation for their Help features.

- **Eclipse** On Unix platforms, the complete online Help is provided in Eclipse format. To access the documentation, run *sadoc* from the *bin32* or *bin64* directory of your SQL Anywhere 11 installation.

- **PDF** The complete set of SQL Anywhere books is provided as a set of Portable Document Format (PDF) files. You must have a PDF reader to view information. To download Adobe Reader, visit Adobe.com.

  To access the PDF documentation on Microsoft Windows operating systems, choose **Start** » **Programs** » **SQL Anywhere 11** » **Documentation** » **Online Books - PDF Format**.

  To access the PDF documentation on Unix operating systems, use a web browser to open *install-dir/documentation/en/pdf/index.html*.

About the books in the documentation set

SQL Anywhere documentation

The SQL Anywhere documentation consists of the following books:

- **SQL Anywhere 11 - Introduction** This book introduces SQL Anywhere 11—a comprehensive package that provides data management and data exchange, enabling the rapid development of database-powered applications for server, desktop, mobile, and remote office environments.
- **SQL Anywhere 11 - Changes and Upgrading** This book describes new features in SQL Anywhere 11 and in previous versions of the software.
- **SQL Anywhere Server - Database Administration** This book describes how to run, manage, and configure SQL Anywhere databases. It describes database connections, the database server, database files, backup procedures, security, high availability, and replication with Replication Server, as well as administration utilities and options.
- **SQL Anywhere Server - Programming** This book describes how to build and deploy database applications using the C, C++, Java, PHP, Perl, Python, and .NET programming languages such as Visual Basic and Visual C#. A variety of programming interfaces such as ADO.NET and ODBC are described.
- **SQL Anywhere Server - SQL Reference** This book provides reference information for system procedures, and the catalog (system tables and views). It also provides an explanation of the SQL Anywhere implementation of the SQL language (search conditions, syntax, data types, and functions).

- **SQL Anywhere Server - SQL Usage** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **MobiLink - Getting Started** This book introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.
- **MobiLink - Client Administration** This book describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases. This book also describes the Dbmlsync API, which allows you to integrate synchronization seamlessly into your C++ or .NET client applications.
- **MobiLink - Server Administration** This book describes how to set up and administer MobiLink applications.
- **MobiLink - Server-Initiated Synchronization** This book describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization or other remote actions from the consolidated database.
- **QAnywhere** This book describes QAnywhere, which is a messaging platform for mobile and wireless clients, as well as traditional desktop and laptop clients.
- **SQL Remote** This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.
- **UltraLite - Database Management and Reference** This book introduces the UltraLite database system for small devices.
- **UltraLite - C and C++ Programming** This book describes UltraLite C and C++ programming interfaces. With UltraLite, you can develop and deploy database applications to handheld, mobile, or embedded devices.
- **UltraLite - M-Business Anywhere Programming** This book describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows Mobile, or Windows.
- **UltraLite - .NET Programming** This book describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.
- **UltraLiteJ** This book describes UltraLiteJ. With UltraLiteJ, you can develop and deploy database applications in environments that support Java. UltraLiteJ supports BlackBerry smartphones and Java SE environments. UltraLiteJ is based on the iAnywhere UltraLite database product.
- **Error Messages** This book provides a complete listing of SQL Anywhere error messages together with diagnostic information.

Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in uppercase, like the words ALTER TABLE in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown like the words *owner* and *table-name* in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like *column-constraint* in the following example:

  ADD *column-definition* [*column-constraint*, ...]

  One or more list elements are allowed. In this example, if more than one is specified, they must be separated by commas.

- **Optional portions** Optional portions of a statement are enclosed by square brackets.

  RELEASE SAVEPOINT [*savepoint-name*]

  These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

- **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  [**ASC** | **DESC**]

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces and a bar is used to separate the options.

  [**QUOTES** { **ON** | **OFF** }]

  If the QUOTES option is used, one of ON or OFF must be provided. The brackets and braces should not be typed.

Operating system conventions

SQL Anywhere runs on a variety of platforms, including Windows, Windows Mobile, Unix, Linux, and Mac OS X. To simplify references to operating systems, the documentation groups the supported operating systems as follows:

- **Windows** The Microsoft Windows family of operating systems for desktop and laptop computers. The Windows family includes Windows Vista and Windows XP.
- **Windows Mobile** Windows Mobile provides a Windows user interface and additional functionality, such as small versions of applications like Word and Excel. Windows Mobile is most commonly used on mobile devices.

Limitations or variations in SQL Anywhere are commonly based on the underlying operating system, and seldom on the particular variant used (Windows Mobile).

- **Unix** Unless specified, Unix refers to Linux, Mac OS X, and Unix platforms.

File name conventions

In many cases, references to file and directory names are similar on all supported platforms, with simple transformations between the various forms. To simplify the documentation in these cases, Windows conventions are used. In other cases, where the details are more complex, the documentation shows all relevant forms.

Here are the conventions used to simplify the documentation of file and directory names:

- **install-dir** During the installation process, you choose where to install SQL Anywhere. The documentation refers to this location using the convention *install-dir*.

  After installation is complete, the environment variable SQLANY11 specifies the location of the installation directory containing the SQL Anywhere components (*install-dir*).

  For example, the documentation may refer to a file as *install-dir\readme.txt*. On Windows platforms, this reference is equivalent to *%SQLANY11%\readme.txt*. On Unix platforms, this reference is equivalent to *$SQLANY11/readme.txt*.

  For more information about the default location of *install-dir*, see "SQLANY11 environment variable" [*SQL Anywhere Server - Database Administration*].

- **Uppercase and lowercase directory names** On Windows, directory names often use mixed case. References to directory names can use any case, since the Windows file system is not case sensitive.

  Unix file systems are case sensitive. The use of mixed-case is less common.

  The SQL Anywhere installation program follows operating system conventions for its directory structure. On Windows, the installation contains directories such as *Bin32* and *Documentation*. On Unix, these directories are called *bin32* and *documentation*.

  The documentation often uses the mixed case forms of directory names. Usually, you can convert a mixed case directory name to lowercase for the equivalent directory name on Unix platforms. For example, the directory *MobiLink* is *mobilink* on Unix platforms.

- **Slashes separating parts of directory names** The documentation uses backslashes as the directory separator. For example, the PDF form of the documentation is found in the directory *install-dir\Documentation\en\pdf*, which is the Windows form.

  On Unix platforms, replace the backslash with the forward slash. The PDF documentation is found in the directory *install-dir/Documentation/en/pdf*.

- **Executable files** The documentation shows executable file names using Windows conventions, with a suffix such as *.exe* or *.bat*. On Unix platforms, executable file names have no suffix.

  For example, on Windows, the network database server is *dbsrv11.exe*. On Unix, Linux, and Mac OS X, it is *dbsrv11*.

- **samples-dir** The installation process allows you to choose where to install the samples that are included with SQL Anywhere, and the documentation refers to this location using the convention *samples-dir*.

  After installation is complete, the environment variable SQLANYSAMP11 specifies the location of the directory containing the samples (*samples-dir*). From the Windows **Start** menu, choosing **Programs** » **SQL Anywhere 11** » **Sample Applications And Projects** opens a Windows Explorer window in this directory.

  For more information about the default location of *samples-dir*, by operating system, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

- **Environment variables** The documentation refers to setting environment variables. On Windows, environment variables are referred to using the syntax *%ENVVAR%*. On Unix, Linux, and Mac OS X, environment variables are referred to using the syntax *$ENVVAR* or *${ENVVAR}*.

  Unix, Linux, and Mac OS X environment variables are stored in shell and login startup files, such as *.cshrc* or *.tcshrc*.

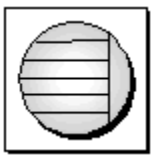
Graphic icons

The following icons are used in this documentation.

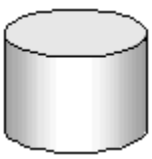
- A client application.

- A database server, such as Sybase SQL Anywhere.

- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.

- Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink server and the SQL Remote Message Agent.

- A programming interface.

Contacting the documentation team

We would like to receive your opinions, suggestions, and feedback on this document.

To submit your comments and suggestions, send an email to the SQL Anywhere documentation team at iasdoc@sybase.com. Although we do not reply to emails, your feedback helps us to improve our documentation, so your input is welcome.

Finding out more and requesting technical support

Additional information and resources, including a code exchange, are available at the Sybase iAnywhere Developer Community at http://www.sybase.com/developer/library/sql-anywhere-techcorner.

If you have questions or need help, you can post messages to the Sybase iAnywhere newsgroups listed below.

When you write to one of these newsgroups, always provide detailed information about your problem, including the build number of your version of SQL Anywhere. You can find this information by running the following command: **dbeng11 -v**.

The newsgroups are located on the *forums.sybase.com* news server. The newsgroups include the following:

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

Newsgroup disclaimer

iAnywhere Solutions has no obligation to provide solutions, information, or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and ensure its operation and availability.

iAnywhere Technical Advisors, as well as other staff, assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.

Part I. Introduction to Programming with SQL Anywhere

This part introduces you to programming with SQL Anywhere,

CHAPTER 1

SQL Anywhere data access programming interfaces

Contents

SQL Anywhere .NET API

ADO.NET is the latest data access API from Microsoft in the line of ODBC, OLE DB, and ADO. It is the preferred data access component for the Microsoft .NET Framework and allows you to access relational database systems.

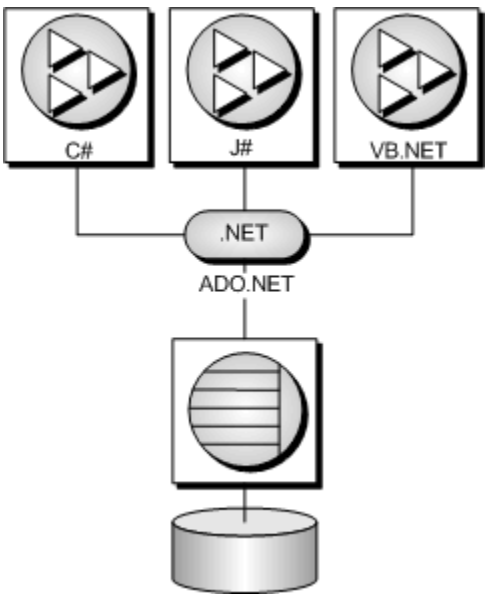
The SQL Anywhere .NET Data Provider implements the iAnywhere.Data.SQLAnywhere namespace and allows you to write programs in any of the .NET supported languages, such as C# and Visual Basic .NET, and access data from SQL Anywhere databases.

For general information about .NET data access, see Microsoft's .NET Data Access Architecture Guide.

ADO.NET applications

You can develop Internet and intranet applications using object-oriented languages, and then connect these applications to SQL Anywhere using the ADO.NET data provider.

Combine this provider with built-in XML and web services features, .NET scripting capability for MobiLink synchronization, and an UltraLite .NET component for development of handheld database applications, and SQL Anywhere is ready to fully integrate with the .NET Framework.



See also

- “SQL Anywhere .NET Data Provider” on page 107
- “iAnywhere.Data.SQLAnywhere namespace (.NET 2.0)” on page 164
- “Tutorial: Using the SQL Anywhere .NET Data Provider” on page 143

SQL Anywhere OLE DB and ADO APIs

SQL Anywhere includes an OLE DB provider for OLE DB and ADO programmers.

OLE DB is a set of Component Object Model (COM) interfaces developed by Microsoft, which provide applications with uniform access to data stored in diverse information sources and that also provide the ability to implement additional database services. These interfaces support the amount of DBMS functionality appropriate to the data store, enabling it to share its data.

ADO is an object model for programmatically accessing, editing, and updating a wide variety of data sources through OLE DB system interfaces. ADO is also developed by Microsoft. Most developers using the OLE DB programming interface do so by writing to the ADO API rather than directly to the OLE DB API.

Do not confuse the ADO interface with ADO.NET. ADO.NET is a separate interface. For more information, see "SQL Anywhere .NET API" on page 4.

Refer to the Microsoft Developer Network for documentation on OLE DB and ADO programming. For SQL Anywhere-specific information about OLE DB and ADO development, see "SQL Anywhere OLE DB and ADO APIs" on page 425.

ODBC API

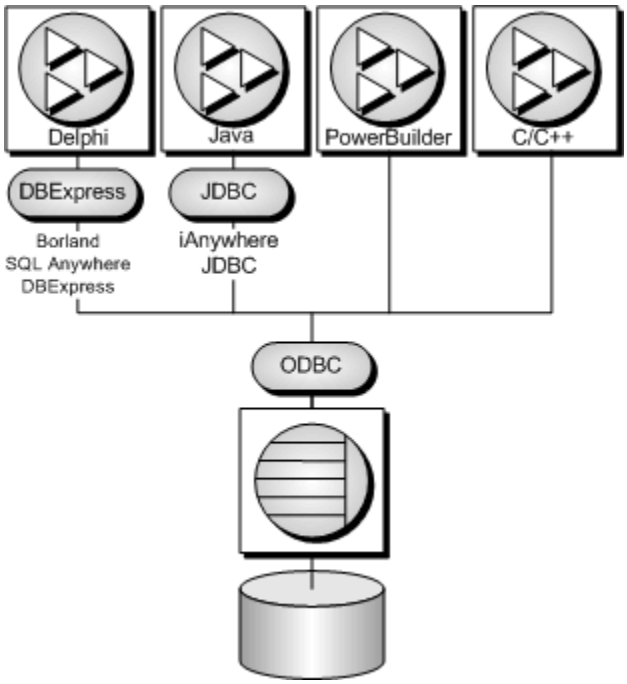
ODBC (Open Database Connectivity) is a standard call level interface (CLI) developed by Microsoft. It is based on the SQL Access Group CLI specification. ODBC applications can run against any data source that provides an ODBC driver. ODBC is a good choice for a programming interface if you want your application to be portable to other data sources that have ODBC drivers.

ODBC is a low-level interface. Almost all the SQL Anywhere functionality is available with this interface. ODBC is available as a DLL under Windows operating systems with the exception of Windows Mobile. It is provided as a library for Unix.

The primary documentation for ODBC is the Microsoft ODBC Software Development Kit.

ODBC applications

You can develop applications using a variety of development tools and programming languages, as shown in the figure below, and accessing the SQL Anywhere database server using the ODBC API.



For example, of the applications supplied with SQL Anywhere, InfoMaker and PowerDesigner Physical Data Model use ODBC to connect to the database.

See also

- "SQL Anywhere ODBC API" on page 441

JDBC API

JDBC is a call-level interface for Java applications. Developed by Sun Microsystems, JDBC provides Java programmers with a uniform interface to a wide range of relational databases, and provides a common base on which higher level tools and interfaces can be built. JDBC is now a standard part of Java and is included in the JDK.

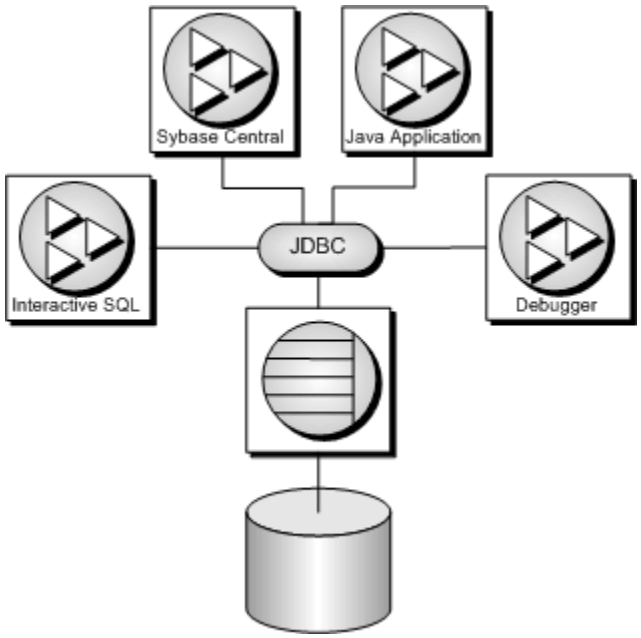
SQL Anywhere includes a pure Java JDBC driver, named jConnect. It also includes the iAnywhere JDBC driver, which is a type 2 driver. Both are described in "SQL Anywhere JDBC API" on page 477.

For information about choosing a driver, see "Choosing a JDBC driver" on page 478.

In addition to using JDBC as a client-side application programming interface, you can also use JDBC inside the database server to access data by using Java in the database.

JDBC applications

You can develop Java applications that use the JDBC API to connect to SQL Anywhere. Several of the applications supplied with SQL Anywhere use JDBC, such as the debugger, Sybase Central, and Interactive SQL.



Java and JDBC are also important programming languages for developing UltraLite applications.

See also

- "Choosing a JDBC driver" on page 478
- "SQL Anywhere JDBC API" on page 477

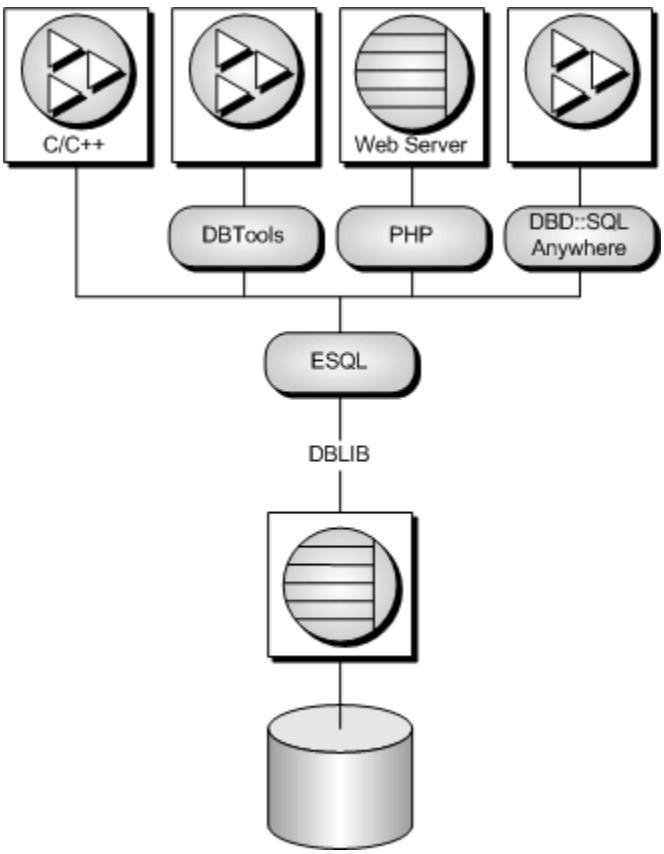
SQL Anywhere embedded SQL

SQL statements embedded in a C or C++ source file are referred to as embedded SQL. A preprocessor translates these statements into calls to a runtime library. Embedded SQL is an ISO/ANSI and IBM standard.

Embedded SQL is portable to other databases and other environments, and is functionally equivalent in all operating environments. It is a comprehensive, low-level interface that provides all of the functionality available in the product. Embedded SQL requires knowledge of C or C++ programming languages.

Embedded SQL applications

You can develop C or C++ applications that access the SQL Anywhere server using the SQL Anywhere embedded SQL interface. The command line database tools are examples of applications developed in this manner.



See also

- "SQL Anywhere embedded SQL" on page 507

SQL Anywhere Perl DBI API

DBD::SQLAnywhere is the SQL Anywhere database driver for DBI, which is a data access API for the Perl language. The DBI API specification defines a set of functions, variables and conventions that provide a consistent database interface independent of the actual database being used. Using DBI and DBD::SQLAnywhere, your Perl scripts have direct access to SQL Anywhere database servers.

See also

- "SQL Anywhere Perl DBD::SQLAnywhere API" on page 607

SQL Anywhere Python Database API

The SQL Anywhere Python database interface, sqlanydb, is a data access API for the Python language. The Python Database API specification defines a set of methods that provides a consistent database interface somewhat independent of the actual database being used. Using the sqlanydb module, your Python scripts have direct access to SQL Anywhere database servers.

See also

- "SQL Anywhere for Python Database API" on page 617

SQL Anywhere PHP API

PHP provides the ability to retrieve information from many popular databases. SQL Anywhere includes a module that provides access to SQL Anywhere databases from PHP. You can use the PHP language to retrieve information from SQL Anywhere databases and provide dynamic web content on your own web sites.

The SQL Anywhere PHP module provides a native means of accessing your databases from PHP. You might prefer it to other PHP data access techniques because it is simple, and it helps to avoid system resource leaks that can occur with other techniques.

See also

- "SQL Anywhere PHP API" on page 625

SQL Anywhere web services

SQL Anywhere web services provide client applications an alternative to data access APIs such as JDBC and ODBC. Web services can be accessed from client applications written in a variety of languages and running on a variety of platforms. Even common scripting languages such as Perl and Python can provide access to web services.

See also

- "SQL Anywhere web services" on page 715

Sybase Open Client API

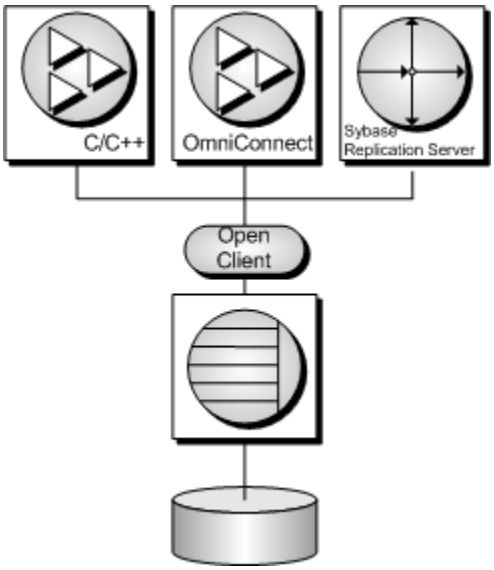
Sybase Open Client provides customer applications, third-party products, and other Sybase products with the interfaces needed to communicate with SQL Anywhere and other Open Servers.

When to use Open Client

You should consider using the Open Client interface if you are concerned with Adaptive Server Enterprise compatibility or if you are using other Sybase products that support the Open Client interface, such as Replication Server.

Open Client applications

You can develop applications in C or C++, and then connect those applications to SQL Anywhere using the Open Client API. Other Sybase applications, such as OmniConnect or Replication Server, use Open Client. The Open Client API is also supported by Sybase Adaptive Server Enterprise.



See also

- "Using SQL Anywhere as an Open Server" [*SQL Anywhere Server - Database Administration*]

CHAPTER 2

SQL Anywhere Explorer

Contents

Introduction to the SQL Anywhere Explorer

The SQL Anywhere Explorer is a component that lets you connect to SQL Anywhere and UltraLite databases from Visual Studio.

For information about using the SQL Anywhere Explorer for UltraLite, see "SQL Anywhere Explorer for UltraLite" [*UltraLite - .NET Programming*].

Using the SQL Anywhere Explorer

In Visual Studio, you can use the SQL Anywhere Explorer to create connections to SQL Anywhere databases. Once you connect to a database, you can:

- view the tables, views, and procedures in the database
- view the data stored in tables and views
- design programs to open connections with the SQL Anywhere database, or to retrieve and manipulate data
- drag and drop database objects onto C# or Visual Basic code or forms so that the IDE automatically generates code that references the selected object

You can also open Sybase Central and Interactive SQL from Visual Studio by choosing the corresponding command from the **Tools** menu.

Installation note

If you install SQL Anywhere software on a Windows computer that already has Visual Studio installed, the installation process detects the presence of Visual Studio and performs the necessary integration steps. If you install Visual Studio after installing SQL Anywhere, or install a new version of Visual Studio, the process to integrate SQL Anywhere with Visual Studio must be performed at a command prompt as follows:

- Ensure Visual Studio is not running.
- At a command prompt, run *install-dir\Assembly\v2\setupVSPackage.exe*.

Working with database connections in Visual Studio

Use the SQL Anywhere Explorer to display the SQL Anywhere database connections under the **Data Connections** node. You must create a data connection to view the data in the tables and views.

You can list database tables, views, stored procedures, and functions in the SQL Anywhere Explorer and expand individual tables to list their columns. The properties for an object selected in the SQL Anywhere Explorer window appear in the Visual Studio **Properties** pane.

To add a SQL Anywhere database connection in Visual Studio

1. Open the SQL Anywhere Explorer by choosing **View** » **SQL Anywhere Explorer**.
2. In the SQL Anywhere Explorer window, right-click **Data Connections**, and then choose **Add Connection**.

   The **Add Connection** window appears.
3. Select **SQL Anywhere**, and then click **OK**.

   The **Connection Properties** window appears.
4. Enter the appropriate values to connect to your database.

5. Click **OK**.

   A connection is made to the database, and the connection is added to the **Data Connections** list.

To remove a SQL Anywhere database connection from Visual Studio

1. Open the SQL Anywhere Explorer by choosing **View** » **SQL Anywhere Explorer**.
2. In the SQL Anywhere Explorer window, right-click the data connections you want to remove, and then choose **Delete**.

   The connection is removed from the SQL Anywhere Explorer window.

Configuring the SQL Anywhere Explorer

The Visual Studio **Options** window includes settings that you can use to configure the SQL Anywhere Explorer.

To access SQL Anywhere Explorer options

1. From the Visual Studio **Tools** menu, choose **Options**.

   The **Options** window appears.
2. In the left pane of the **Options** window, expand **SQL Anywhere**.
3. Click **General** to configure the SQL Anywhere Explorer general options as required.

   Limit Query Results Sent To Output Window Specify the number of rows that appear in the **Output** window. The default value is 500.

   Show System Objects In Server Explorer Check this option if you want to see system objects in the Microsoft Server Explorer. This is not a SQL Anywhere Explorer option but a Server Explorer option. System objects include those owned by the "dbo" user.

   Sort Objects Choose to sort objects in the SQL Anywhere Explorer window by object name or by object owner name.

   Generate UI Code When Dropping A Table Or View Onto The Designer Generate the code for tables or views that you drag and drop onto the Windows forms designer.

   Generate Insert, Update, And Delete Commands For Data Adapters Generate INSERT, UPDATE, and DELETE commands for the data adapter when you drag and drop a table or view onto a C# or Visual Basic document.

   Generate Table Mappings For Data Adapters Generate table mappings for the data adapter when you drag and drop a table onto a C# or Visual Basic document.

Adding database objects using the SQL Anywhere Explorer

In Visual Studio, when you drag certain database objects from the SQL Anywhere Explorer and drop them onto Visual Studio designers, the IDE automatically creates new components that reference the selected

objects. You can configure the settings for drag and drop operations by choosing **Tools** » **Options** in Visual Studio and opening the SQL Anywhere node.

For example, if you drag a stored procedure from the SQL Anywhere Explorer onto a Windows form, the IDE automatically creates a Command object preconfigured to call that stored procedure.

The following table lists the objects you can drag from the SQL Anywhere Explorer, and describes the components created when you drop them onto a Visual Studio Forms Designer or Code Editor.

Item	Result
Data connection	Creates a data connection.
Table	Creates an adapter.
View	Creates an adapter.
Stored procedure or function	Creates a command.

To create a new data component using the SQL Anywhere Explorer

1. Open the form or class that you want to add a data component to.
2. In the SQL Anywhere Explorer, select the object you want to use.
3. Drag the object from the SQL Anywhere Explorer to the Forms Designer or Code Editor.

Working with tables using the SQL Anywhere Explorer

The SQL Anywhere Explorer enables you to view the properties and data for tables and views in a SQL Anywhere database from within Visual Studio.

To view a table or view data in Visual Studio

1. Connect to a SQL Anywhere database using the SQL Anywhere Explorer.
2. In the SQL Anywhere Explorer window, expand your database, and then expand Tables or Views, depending on the object you want to view.
3. Right-click a table or view, and then choose **Retrieve Data**.

   The data in the selected table or view appears in the **Output** window in Visual Studio.

Working with procedures and functions using the SQL Anywhere Explorer

If changes are made to a stored procedure, you can refresh the procedure from the SQL Anywhere Explorer to get the latest changes to columns or parameters.

To refresh a procedure in Visual Studio

1. Connect to a SQL Anywhere database.
2. Right-click the procedure and choose **Refresh**.

   The parameters and columns are updated if any changes have been made to the procedure in the database.

CHAPTER 3

Using SQL in applications

Contents

Executing SQL statements in applications

The way you include SQL statements in your application depends on the application development tool and programming interface you use.

- **ADO.NET** You can execute SQL statements using a variety of ADO.NET objects. The SACommand object is one example:

```
SACommand cmd = new SACommand(
     "DELETE FROM Employees WHERE EmployeeID = 105", conn );
cmd.ExecuteNonQuery();
```

  See "SQL Anywhere .NET Data Provider" on page 107.

- **ODBC** If you are writing directly to the ODBC programming interface, your SQL statements appear in function calls. For example, the following C function call executes a DELETE statement:

```
SQLExecDirect( stmt,
    "DELETE FROM Employees
     WHERE EmployeeID = 105",
    SQL_NTS );
```

  See "SQL Anywhere ODBC API" on page 441.

- **JDBC** If you are using the JDBC programming interface, you can execute SQL statements by invoking methods of the statement object. For example:

```
stmt.executeUpdate(
    "DELETE FROM Employees
     WHERE EmployeeID = 105" );
```

  See "SQL Anywhere JDBC API" on page 477.

- **Embedded SQL** If you are using embedded SQL, you prefix your C language SQL statements with the keyword EXEC SQL. The code is then run through a preprocessor before compiling. For example:

```
EXEC SQL EXECUTE IMMEDIATE
 'DELETE FROM Employees
  WHERE EmployeeID = 105';
```

  See "SQL Anywhere embedded SQL" on page 507.

- **Sybase Open Client** If you use the Sybase Open Client interface, your SQL statements appear in function calls. For example, the following pair of calls executes a DELETE statement:

```
ret = ct_command( cmd, CS_LANG_CMD,
                  "DELETE FROM Employees
                   WHERE EmployeeID=105"
                 CS_NULLTERM,
                 CS_UNUSED);
ret = ct_send(cmd);
```

  See "Sybase Open Client API" on page 705.

- **Application development tools** Application development tools such as the members of the Sybase Enterprise Application Studio family provide their own SQL objects, which use either ODBC (PowerBuilder) or JDBC (Power J) under the covers.

For more detailed information about how to include SQL in your application, see your development tool documentation. If you are using ODBC or JDBC, consult the software development kit for those interfaces.

Applications inside the database server

In many ways, stored procedures and triggers act as applications or parts of applications running inside the database server. You can also use many of the techniques here in stored procedures.

For more information about stored procedures and triggers, see "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*].

Java classes in the database can use the JDBC interface in the same way as Java applications outside the server. This chapter discusses some aspects of JDBC. For more information about using JDBC, see "SQL Anywhere JDBC API" on page 477.

Preparing statements

Each time a statement is sent to a database, the database server must perform the following steps:

- It must parse the statement and transform it into an internal form. This is sometimes called **preparing** the statement.
- It must verify the correctness of all references to database objects by checking, for example, that columns named in a query actually exist.
- The query optimizer generates an access plan if the statement involves joins or subqueries.
- It executes the statement after all these steps have been carried out.

Reusing prepared statements can improve performance

If you find yourself using the same statement repeatedly, for example inserting many rows into a table, repeatedly preparing the statement causes a significant and unnecessary overhead. To remove this overhead, some database programming interfaces provide ways of using prepared statements. A **prepared statement** is a statement containing a series of placeholders. When you want to execute the statement, all you have to do is assign values to the placeholders, rather than prepare the entire statement over again.

Using prepared statements is particularly useful when carrying out many similar actions, such as inserting many rows.

Generally, using prepared statements requires the following steps:

1. **Prepare the statement**

   In this step you generally provide the statement with some placeholder character instead of the values.

2. **Repeatedly execute the prepared statement**

   In this step you supply values to be used each time the statement is executed. The statement does not have to be prepared each time.

3. **Drop the statement**

   In this step you free the resources associated with the prepared statement. Some programming interfaces handle this step automatically.

Do not prepare statements that are used only once

In general, you should not prepare statements if they are only executed once. There is a slight performance penalty for separate preparation and execution, and it introduces unnecessary complexity into your application.

In some interfaces, however, you do need to prepare a statement to associate it with a cursor.

For information about cursors, see “Introduction to cursors” on page 27.

The calls for preparing and executing statements are not a part of SQL, and they differ from interface to interface. Each of the SQL Anywhere programming interfaces provides a method for using prepared statements.

How to use prepared statements

This section provides a brief overview of how to use prepared statements. The general procedure is the same, but the details vary from interface to interface. Comparing how to use prepared statements in different interfaces illustrates this point.

To use a prepared statement (generic)

1. Prepare the statement.
2. Bind the parameters that will hold values in the statement.
3. Assign values to the bound parameters in the statement.
4. Execute the statement.
5. Repeat steps 3 and 4 as needed.
6. Drop the statement when finished. This step is not required in JDBC, as Java's garbage collection mechanisms handle this for you.

To use a prepared statement (ADO.NET)

1. Create an SACommand object holding the statement.

   ```
   SACommand cmd = new SACommand(
        "SELECT * FROM Employees WHERE Surname=?", conn );
   ```

2. Declare data types for any parameters in the statement.

   Use the SACommand.CreateParameter method.

3. Prepare the statement using the Prepare method.

   ```
   cmd.Prepare();
   ```

4. Execute the statement.

   ```
   SADataReader reader = cmd.ExecuteReader();
   ```

   For an example of preparing statements using ADO.NET, see the source code in *samples-dir\SQLAnywhere\ADO.NET\SimpleWin32*.

To use a prepared statement (ODBC)

1. Prepare the statement using SQLPrepare.
2. Bind the statement parameters using SQLBindParameter.
3. Execute the statement using SQLExecute.
4. Drop the statement using SQLFreeStmt.

   For an example of preparing statements using ODBC, see the source code in *samples-dir\SQLAnywhere\ODBCPrepare*.

   For more information about ODBC prepared statements, see the ODBC SDK documentation, and "Executing prepared statements" on page 460.

To use a prepared statement (JDBC)

1. Prepare the statement using the prepareStatement method of the connection object. This returns a prepared statement object.
2. Set the statement parameters using the appropriate **set***Type* methods of the prepared statement object. Here, *Type* is the data type assigned.
3. Execute the statement using the appropriate method of the prepared statement object. For inserts, updates, and deletes this is the executeUpdate method.

   For an example of preparing statements using JDBC, see the source code file *samples-dir\SQLAnywhere \JDBC\JDBCExample.java*.

   For more information about using prepared statements in JDBC, see "Using prepared statements for more efficient access" on page 495.

To use a prepared statement (embedded SQL)

1. Prepare the statement using the EXEC SQL PREPARE statement.
2. Assign values to the parameters in the statement.
3. Execute the statement using the EXEC SQL EXECUTE statement.
4. Free the resources associated with the statement using the EXEC SQL DROP statement.

   For more information about embedded SQL prepared statements, see "PREPARE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

To use a prepared statement (Open Client)

1. Prepare the statement using the ct_dynamic function, with a CS_PREPARE type parameter.
2. Set statement parameters using ct_param.
3. Execute the statement using ct_dynamic with a CS_EXECUTE type parameter.
4. Free the resources associated with the statement using ct_dynamic with a CS_DEALLOC type parameter.

   For more information about using prepared statements in Open Client, see "Using SQL in Open Client applications" on page 710.

Introduction to cursors

When you execute a query in an application, the result set consists of a number of rows. In general, you do not know how many rows the application is going to receive before you execute the query. Cursors provide a way of handling query result sets in applications.

The way you use cursors and the kinds of cursors available to you depend on the programming interface you use. For a list of cursor types available from each interface, see "Availability of cursors" on page 37.

With cursors, you can perform the following tasks within any programming interface:

- Loop over the results of a query.
- Perform inserts, updates, and deletes on the underlying data at any point within a result set.

In addition, some programming interfaces allow you to use special features to tune the way result sets return to your application, providing substantial performance benefits for your application.

For more information about the kinds of cursors available through different programming interfaces, see "Availability of cursors" on page 37.

What are cursors?

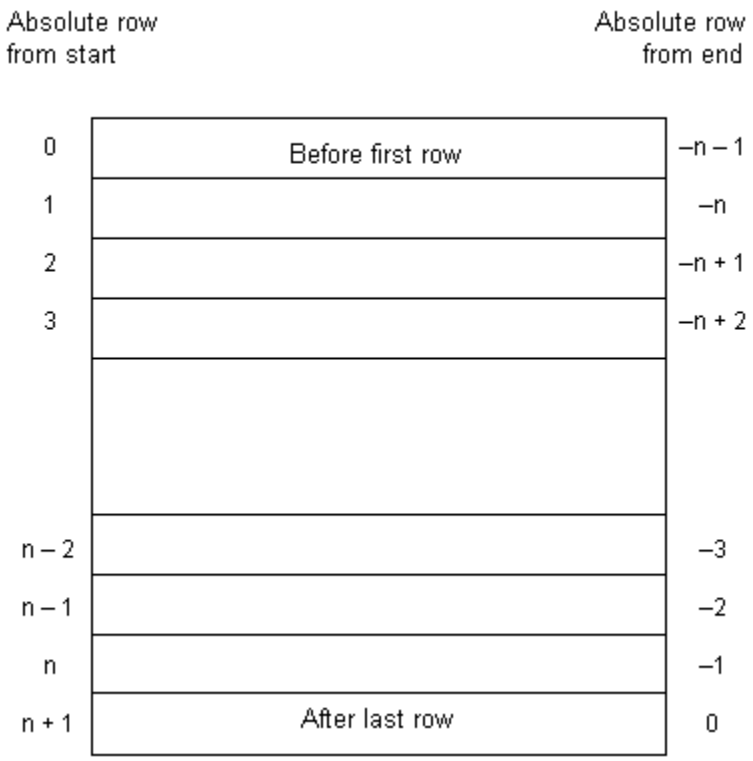
A **cursor** is a name associated with a result set. The result set is obtained from a SELECT statement or stored procedure call.

A cursor is a handle on the result set. At any time, the cursor has a well-defined position within the result set. With a cursor you can examine and possibly manipulate the data one row at a time. SQL Anywhere cursors support forward and backward movement through the query results.

Cursor positions

Cursors can be positioned in the following places:

- Before the first row of the result set.
- On a row in the result set.
- After the last row of the result set.



The cursor position and result set are maintained in the database server. Rows are **fetched** by the client for display and processing either one at a time or a few at a time. The entire result set does not need to be delivered to the client.

Benefits of using cursors

You do not need to use cursors in database applications, but they do provide a number of benefits. These benefits follow from the fact that if you do not use a cursor, the entire result set must be transferred to the client for processing and display:

- **Client-side memory** For large results, holding the entire result set on the client can lead to demanding memory requirements.
- **Response time** Cursors can provide the first few rows before the whole result set is assembled. If you do not use cursors, the entire result set must be delivered before any rows are displayed by your application.
- **Concurrency control** If you make updates to your data and do not use cursors in your application, you must send separate SQL statements to the database server to apply the changes. This raises the possibility of concurrency problems if the result set has changed since it was queried by the client. In turn, this raises the possibility of lost updates.

Cursors act as pointers to the underlying data, and so impose proper concurrency constraints on any changes you make.

Working with cursors

This section describes how to perform different kinds of operations using cursors.

Using cursors

Using a cursor in embedded SQL is different than using a cursor in other interfaces.

To use a cursor (ADO.NET, ODBC, JDBC, and Open Client)

1. Prepare and execute a statement.

   Execute a statement using the usual method for the interface. You can prepare and then execute the statement, or you can execute the statement directly.

   With ADO.NET, only the SACommand.ExecuteReader method returns a cursor. It provides a read-only, forward-only cursor.

2. Test to see if the statement returns a result set.

   A cursor is implicitly opened when a statement that creates a result set is executed. When the cursor is opened, it is positioned before the first row of the result set.

3. Fetch results.

   Although simple fetch operations move the cursor to the next row in the result set, SQL Anywhere permits more complicated movement around the result set.

4. Close the cursor.

   When you have finished with the cursor, close it to free associated resources.

5. Free the statement.

   If you used a prepared statement, free it to reclaim memory.

To use a cursor (embedded SQL)

1. Prepare a statement.

   Cursors generally use a statement handle rather than a string. You need to prepare a statement to have a handle available.

   For information about preparing a statement, see "Preparing statements" on page 24.

2. Declare the cursor.

   Each cursor refers to a single SELECT or CALL statement. When you declare a cursor, you state the name of the cursor and the statement it refers to.

   For more information, see "DECLARE CURSOR statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].

3. Open the cursor.

For more information, see "OPEN statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].

In the case of a CALL statement, opening the cursor executes the procedure up to the point where the first row is about to be obtained.

4. Fetch results.

   Although simple fetch operations move the cursor to the next row in the result set, SQL Anywhere permits more complicated movement around the result set. How you declare the cursor determines which fetch operations are available to you.

   For more information, see "FETCH statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*], and "Fetching data" on page 550.

5. Close the cursor.

   When you have finished with the cursor, close it. This frees any resources associated with the cursor.

   For more information, see "CLOSE statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].

6. Drop the statement.

   To free the memory associated with the statement, you must drop the statement.

   For more information, see "DROP STATEMENT statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

For more information about using cursors in embedded SQL, see "Fetching data" on page 550.

Prefetching rows

In some cases, the interface library may undertake performance optimizations under the covers (such as prefetching results), so these steps in the client application may not correspond exactly to software operations.

Cursor positioning

When a cursor is opened, it is positioned before the first row. You can move the cursor position to an absolute position from the start or the end of the query results, or to a position relative to the current cursor position. The specifics of how you change cursor position, and what operations are possible, are governed by the programming interface.

The number of row positions you can fetch in a cursor is governed by the size of an integer. You can fetch rows numbered up to number 2147483646, which is one less than the value that can be held in an integer. When using negative numbers (rows from the end) you can fetch down to one more than the largest negative value that can be held in an integer.

You can use special positioned update and delete operations to update or delete the row at the current position of the cursor. If the cursor is positioned before the first row or after the last row, an error is returned indicating that there is no corresponding cursor row.

Cursor positioning problems
Inserts and some updates to asensitive cursors can cause problems with cursor positioning. SQL Anywhere does not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row does not appear at all until the cursor is closed and opened again. With SQL Anywhere, this occurs if a work table had to be created to open the cursor (see "Use work tables in query processing (use All-rows optimization goal)" [*SQL Anywhere Server - SQL Usage*] for a description).

The UPDATE statement may cause a row to move in the cursor. This happens if the cursor has an ORDER BY clause that uses an existing index (a work table is not created). Using STATIC SCROLL cursors alleviates these problems but requires more memory and processing.

Configuring cursors on opening

You can configure the following aspects of cursor behavior when you open the cursor:

- **Isolation level** You can explicitly set the isolation level of operations on a cursor to be different from the current isolation level of the transaction. To do this, set the isolation_level option.

  For more information, see "isolation_level option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **Holding** By default, cursors in embedded SQL close at the end of a transaction. Opening a cursor WITH HOLD allows you to keep it open until the end of a connection, or until you explicitly close it. ADO.NET, ODBC, JDBC, and Open Client leave cursors open at the end of transactions by default.

Fetching rows through a cursor

The simplest way of processing the result set of a query using a cursor is to loop through all the rows of the result set until there are no more rows.

To loop through the rows of a result set

1. Declare and open the cursor (embedded SQL), or execute a statement that returns a result set (ODBC, JDBC, Open Client) or SADataReader object (ADO.NET).
2. Continue to fetch the next row until you get a `Row Not Found` error.
3. Close the cursor.

How step 2 of this operation is carried out depends on the interface you use. For example,

- **ADO.NET** Use the SADataReader.NextResult method. See "NextResult method" on page 318.
- **ODBC** SQLFetch, SQLExtendedFetch, or SQLFetchScroll advances the cursor to the next row and returns the data.

  For more information about using cursors in ODBC, see "Working with result sets" on page 462.

- **JDBC** The next method of the ResultSet object advances the cursor and returns the data.

  For more information about using the ResultSet object in JDBC, see "Returning result sets" on page 499.

- **Embedded SQL** The FETCH statement carries out the same operation.

  For more information about using cursors in embedded SQL, see "Using cursors in embedded SQL" on page 551.

- **Open Client** The ct_fetch function advances the cursor to the next row and returns the data.

  For more information about using cursors in Open Client applications, see "Using cursors" on page 710.

Fetching multiple rows

Multiple-row fetching should not be confused with prefetching rows. Multiple row fetching is performed by the application, while prefetching is transparent to the application, and provides a similar performance gain. Fetching multiple rows at a time can improve performance.

Multiple-row fetches

Some interfaces provide methods for fetching more than one row at a time into the next several fields in an array. Generally, the fewer separate fetch operations you execute, the fewer individual requests the server must respond to, and the better the performance. A modified FETCH statement that retrieves multiple rows is also sometimes called a **wide fetch**. Cursors that use multiple-row fetches are sometimes called **block cursors** or **fat cursors**.

Using multiple-row fetching

- In ODBC, you can set the number of rows that will be returned on each call to SQLFetchScroll or SQLExtendedFetch by setting the SQL_ATTR_ROW_ARRAY_SIZE or SQL_ROWSET_SIZE attribute.
- In embedded SQL, the FETCH statement uses an ARRAY clause to control the number of rows fetched at a time.
- Open Client and JDBC do not support multi-row fetches. They do use prefetching.

Fetching with scrollable cursors

ODBC and embedded SQL provide methods for using scrollable cursors and dynamic scrollable cursors. These methods allow you to move several rows forward at a time, or to move backward through the result set.

The JDBC and Open Client interfaces do not support scrollable cursors.

Prefetching does not apply to scrollable operations. For example, fetching a row in the reverse direction does not prefetch several previous rows.

Modifying rows through a cursor

Cursors can do more than just read result sets from a query. You can also modify data in the database while processing a cursor. These operations are commonly called **positioned** insert, update, and delete operations, or **PUT** operations if the action is an insert.

Not all query result sets allow positioned updates and deletes. If you perform a query on a non-updatable view, then no changes occur to the underlying tables. Also, if the query involves a join, then you must specify which table you want to delete from, or which columns you want to update, when you perform the operations.

Inserts through a cursor can only be executed if any non-inserted columns in the table allow NULL or have defaults.

If multiple rows are inserted into a value-sensitive (keyset driven) cursor, they appear at the end of the cursor result set. The rows appear at the end, even if they do not match the WHERE clause of the query or if an ORDER BY clause would normally have placed them at another location in the result set. This behavior is independent of programming interface. For example, it applies when using the embedded SQL PUT statement or the ODBC SQLBulkOperations function. The value of an autoincrement column for the most recent row inserted can be found by selecting the last row in the cursor. For example, in embedded SQL the value could be obtained using `FETCH ABSOLUTE -1 cursor-name`. As a result of this behavior, the first multiple-row insert for a value-sensitive cursor may be expensive.

ODBC, JDBC, embedded SQL, and Open Client permit data modification using cursors, but ADO.NET does not. With Open Client, you can delete and update rows, but you can only insert rows on a single-table query.

Which table are rows deleted from?

If you attempt a positioned delete through a cursor, the table from which rows are deleted is determined as follows:

1. If no FROM clause is included in the DELETE statement, the cursor must be on a single table only.
2. If the cursor is for a joined query (including using a view containing a join), then the FROM clause must be used. Only the current row of the specified table is deleted. The other tables involved in the join are not affected.
3. If a FROM clause is included, and no table owner is specified, the table-spec value is first matched against any correlation names.

   For more information, see "FROM clause" [*SQL Anywhere Server - SQL Reference*].
4. If a correlation name exists, the table-spec value is identified with the correlation name.
5. If a correlation name does not exist, the table-spec value must be unambiguously identifiable as a table name in the cursor.
6. If a FROM clause is included, and a table owner is specified, the table-spec value must be unambiguously identifiable as a table name in the cursor.
7. The positioned DELETE statement can be used on a cursor open on a view as long as the view is updatable.

Understanding updatable statements

This section describes how clauses in the SELECT statement affect updatable statements and cursors.

Updatability of read-only statements

Specifying FOR READ ONLY in the cursor declaration, or including a FOR READ ONLY clause in the statement, renders the statement read-only. In other words, a FOR READ ONLY clause, or the appropriate read-only cursor declaration when using a client API, overrides any other updatability specification.

If the outermost block of a SELECT statement contains an ORDER BY clause, and the statement does not specify FOR UPDATE, then the cursor is READ ONLY. If the SQL SELECT statement specifies FOR XML, then the cursor is READ ONLY. Otherwise, the cursor is updatable.

Updatable statements and concurrency control

For updatable statements, SQL Anywhere provides both optimistic and pessimistic concurrency control mechanisms on cursors to ensure that a result set remains consistent during scrolling operations. These mechanisms are alternatives to using INSENSITIVE cursors or snapshot isolation, although they have different semantics and tradeoffs.

The specification of FOR UPDATE can affect whether or not a cursor is updatable, but in SQL Anywhere, the FOR UPDATE syntax alone has no other effect on concurrency control. If FOR UPDATE is specified with additional parameters, SQL Anywhere alters the processing of the statement to incorporate one of two concurrency control options as follows:

- **Pessimistic** For all rows fetched in the cursor's result set, the database server acquires intent row locks to prevent the rows from being updated by any other transaction.
- **Optimistic** The cursor type used by the database server is changed to a keyset-driven cursor (insensitive row membership, value-sensitive) so that the application can be informed when a row in the result has been modified or deleted by this, or any other transaction.

Pessimistic or optimistic concurrency is specified at the cursor level either through options with DECLARE CURSOR or FOR statements, or though the concurrency setting API for a specific programming interface. If a statement is updatable and the cursor does not specify a concurrency control mechanism, the statement's specification is used. The syntax is as follows:

- **FOR UPDATE BY LOCK** The database server acquires intent row locks on fetched rows of the result set. These are long-term locks that are held until transaction COMMIT or ROLLBACK.
- **FOR UPDATE BY { VALUES | TIMESTAMP }** The database server utilizes a keyset-driven cursor to enable the application to be informed when rows have been modified or deleted as the result set is scrolled.

For more information, see "DECLARE statement" [*SQL Anywhere Server - SQL Reference*], and "FOR statement" [*SQL Anywhere Server - SQL Reference*].

Restricting updatable statements

FOR UPDATE (*column-list*) enforces the restriction that only named result set attributes can be modified in a subsequent UPDATE WHERE CURRENT OF statement.

Canceling cursor operations

You can cancel a request through an interface function. From Interactive SQL, you can cancel a request by clicking Interrupt SQL Statement on the toolbar (or by choosing Stop from the SQL menu).

If you cancel a request that is carrying out a cursor operation, the position of the cursor is indeterminate. After canceling the request, you must locate the cursor by its absolute position, or close it.

Choosing cursor types

This section describes mappings between SQL Anywhere cursors and the options available to you from the programming interfaces supported by SQL Anywhere.

For information about SQL Anywhere cursors, see "SQL Anywhere cursors" on page 39.

Availability of cursors

Not all interfaces provide support for all types of cursors.

- ADO.NET provides only forward-only, read-only cursors.
- ADO/OLE DB and ODBC support all types of cursors.

  For more information, see "Working with result sets" on page 462.
- Embedded SQL supports all types of cursors.
- For JDBC:
  - The iAnywhere JDBC driver supports the JDBC 2.0 and JDBC 3.0 specifications and permits the declaration of insensitive, sensitive, and forward-only asensitive cursors.
  - jConnect 5.5 and 6.0.5 support the declaration of insensitive, sensitive, and forward-only asensitive cursors in the same manner as the iAnywhere JDBC driver. However, the underlying implementation of jConnect only supports asensitive cursor semantics.

    For more information about declaring JDBC cursors, see "Requesting SQL Anywhere cursors" on page 53.
- Sybase Open Client supports only asensitive cursors. Also, a severe performance penalty results when using updatable, non-unique cursors.

Cursor properties

You request a cursor type, either explicitly or implicitly, from the programming interface. Different interface libraries offer different choices of cursor types. For example, JDBC and ODBC specify different cursor types.

Each cursor type is defined by a number of characteristics:

- **Uniqueness** Declaring a cursor to be unique forces the query to return all the columns required to uniquely identify each row. Often this means returning all the columns in the primary key. Any columns required but not specified are added to the result set. The default cursor type is non-unique.
- **Updatability** A cursor declared as read only cannot be used in a positioned update or delete operation. The default cursor type is updatable.
- **Scrollability** You can declare cursors to behave different ways as you move through the result set. Some cursors can fetch only the current row or the following row. Others can move backward and forward through the result set.

- **Sensitivity** Changes to the database may or may not be visible through a cursor.

These characteristics may have significant side effects on performance and on database server memory usage.

SQL Anywhere makes available cursors with a variety of mixes of these characteristics. When you request a cursor of a given type, SQL Anywhere matches those characteristics as well as it can.

There are some occasions when not all characteristics can be supplied. For example, insensitive cursors in SQL Anywhere must be read-only. If your application requests an updatable insensitive cursor, a different cursor type (value-sensitive) is supplied instead.

Bookmarks and cursors

ODBC provides **bookmarks**, or values, used to identify rows in a cursor. SQL Anywhere supports bookmarks for value-sensitive and insensitive cursors. For example, this means that the ODBC cursor types SQL_CURSOR_STATIC and SQL_CURSOR_KEYSET_DRIVEN support bookmarks while cursor types SQL_CURSOR_DYNAMIC and SQL_CURSOR_FORWARD_ONLY do not.

Block cursors

ODBC provides a cursor type called a block cursor. When you use a BLOCK cursor, you can use SQLFetchScroll or SQLExtendedFetch to fetch a block of rows, rather than a single row. Block cursors behave identically to embedded SQL ARRAY fetches.

SQL Anywhere cursors

Any cursor, once opened, has an associated result set. The cursor is kept open for a length of time. During that time, the result set associated with the cursor may be changed, either through the cursor itself or, subject to isolation level requirements, by other transactions. Some cursors permit changes to the underlying data to be visible, while others do not reflect these changes. The different behavior of cursors with respect to changes to the underlying data is the **sensitivity** of the cursor.

SQL Anywhere provides cursors with a variety of sensitivity characteristics. This section describes what sensitivity is, and describes the sensitivity characteristics of cursors.

This section assumes that you have read "What are cursors?" on page 27.

Membership, order, and value changes

Changes to the underlying data can affect the result set of a cursor in the following ways:

- **Membership** The set of rows in the result set, as identified by their primary key values.
- **Order** The order of the rows in the result set.
- **Value** The values of the rows in the result set.

For example, consider the following simple table with employee information (EmployeeID is the primary key column):

EmployeeID	Surname
1	Whitney
2	Cobb
3	Chin

A cursor on the following query returns all results from the table in primary key order:

```
SELECT EmployeeID, Surname
FROM Employees
ORDER BY EmployeeID;
```

The membership of the result set could be changed by adding a new row or deleting a row. The values could be changed by changing one of the names in the table. The order could be changed by changing the primary key value of one of the employees.

Visible and invisible changes

Subject to isolation level requirements, the membership, order, and values of the result set of a cursor can be changed after the cursor is opened. Depending on the type of cursor in use, the result set as seen by the application may or may not change to reflect these changes.

Changes to the underlying data may be **visible** or **invisible** through the cursor. A visible change is a change that is reflected in the result set of the cursor. Changes to the underlying data that are not reflected in the result set seen by the cursor are invisible.

Cursor sensitivity overview

SQL Anywhere cursors are classified by their sensitivity with respect to changes of the underlying data. In particular, cursor sensitivity is defined in terms of which changes are visible.

- **Insensitive cursors** The result set is fixed when the cursor is opened. No changes to the underlying data are visible. See "Insensitive cursors" on page 44.
- **Sensitive cursors** The result set can change after the cursor is opened. All changes to the underlying data are visible. See "Sensitive cursors" on page 44.
- **Asensitive cursors** Changes may be reflected in the membership, order, or values of the result set seen through the cursor, or may not be reflected at all. See "Asensitive cursors" on page 46.
- **Value-sensitive cursors** Changes to the order or values of the underlying data are visible. The membership of the result set is fixed when the cursor is opened. See "Value-sensitive cursors" on page 47.

The differing requirements on cursors place different constraints on execution, and so, performance. For more information, see "Cursor sensitivity and performance" on page 48.

Cursor sensitivity example: A deleted row

This example uses a simple query to illustrate how different cursors respond to a row in the result set being deleted.

Consider the following sequence of events:

1. An application opens a cursor on the following query against the sample database.

```
SELECT EmployeeID, Surname
FROM Employees
ORDER BY EmployeeID;
```

EmployeeID	Surname
102	Whitney
105	Cobb
160	Breault
...	...

2. The application fetches the first row through the cursor (102).
3. The application fetches the next row through the cursor (105).
4. A separate transaction deletes employee 102 (Whitney) and commits the change.

The results of cursor actions in this situation depend on the cursor sensitivity:

- **Insensitive cursors** The DELETE is not reflected in either the membership or values of the results as seen through the cursor:

Action	Result
Fetch previous row	Returns the original copy of the row (102).
Fetch the first row (absolute fetch)	Returns the original copy of the row (102).
Fetch the second row (absolute fetch)	Returns the unchanged row (105).

- **Sensitive cursors** The membership of the result set has changed so that row 105 is now the first row in the result set:

Action	Result
Fetch previous row	Returns `Row Not Found`. There is no previous row.
Fetch the first row (absolute fetch)	Returns row 105.
Fetch the second row (absolute fetch)	Returns row 160.

- **Value-sensitive cursors** The membership of the result set is fixed, and so row 105 is still the second row of the result set. The DELETE is reflected in the values of the cursor, and creates an effective hole in the result set.

Action	Result
Fetch previous row	Returns `No current row of cursor`. There is a hole in the cursor where the first row used to be.
Fetch the first row (absolute fetch)	Returns `No current row of cursor`. There is a hole in the cursor where the first row used to be.
Fetch the second row (absolute fetch)	Returns row 105.

- **Asensitive cursors** The membership and values of the result set are indeterminate with respect to the changes. The response to a fetch of the previous row, the first row, or the second row depends on the particular optimization method for the query, whether that method involved the formation of a work table, and whether the row being fetched was prefetched from the client.

  The benefit of asensitive cursors is that for many applications, sensitivity is unimportant. In particular, if you are using a forward-only, read-only cursor, no underlying changes are seen. Also, if you are running at a high isolation level, underlying changes are disallowed.

Cursor sensitivity example: An updated row

This example uses a simple query to illustrate how different cursor types respond to a row in the result set being updated in such a way as to change the order of the result set.

Consider the following sequence of events:

1. An application opens a cursor on the following query against the sample database.

```
SELECT EmployeeID, Surname
FROM Employees;
```

EmployeeID	Surname
102	Whitney
105	Cobb
160	Breault
...	...

2. The application fetches the first row through the cursor (102).
3. The application fetches the next row through the cursor (105).
4. A separate transaction updates the employee ID of employee 102 (Whitney) to 165 and commits the change.

The results of the cursor actions in this situation depend on the cursor sensitivity:

- **Insensitive cursors** The UPDATE is not reflected in either the membership or values of the results as seen through the cursor:

Action	Result
Fetch previous row	Returns the original copy of the row (102).
Fetch the first row (absolute fetch)	Returns the original copy of the row (102).
Fetch the second row (absolute fetch)	Returns the unchanged row (105).

- **Sensitive cursors** The membership of the result set has changed so that row 105 is now the first row in the result set:

Action	Result
Fetch previous row	Returns `Row Not Found`. The membership of the result set has changed so that 105 is now the first row. The cursor is moved to the position before the first row.

Action	Result
Fetch the first row (absolute fetch)	Returns row 105.
Fetch the second row (absolute fetch)	Returns row 160.

In addition, a fetch on a sensitive cursor returns the warning `SQLE_ROW_UPDATED_WARNING` if the row has changed since the last reading. The warning is given only once. Subsequent fetches of the same row do not produce the warning.

Similarly, a positioned update or delete through the cursor on a row since it was last fetched returns the SQLE_ROW_UPDATED_SINCE_READ error. An application must fetch the row again for an update or delete on a sensitive cursor to work.

An update to any column causes the warning/error, even if the column is not referenced by the cursor. For example, a cursor on a query returning Surname would report the update even if only the Salary column was modified.

- **Value-sensitive cursors** The membership of the result set is fixed, and so row 105 is still the second row of the result set. The UPDATE is reflected in the values of the cursor, and creates an effective "hole" in the result set.

Action	Result
Fetch previous row	Returns `Row Not Found`. The membership of the result set has changed so that 105 is now the first row: The cursor is positioned on the hole: it is before row 105.
Fetch the first row (absolute fetch)	Returns `No current row of cursor`. The membership of the result set has changed so that 105 is now the first row: The cursor is positioned on the hole: it is before row 105.
Fetch the second row (absolute fetch)	Returns row 105.

- **Asensitive cursors** The membership and values of the result set are indeterminate with respect to the changes. The response to a fetch of the previous row, the first row, or the second row depends on the particular optimization method for the query, whether that method involved the formation of a work table, and whether the row being fetched was prefetched from the client.

No warnings or errors in bulk operations mode
Update warning and error conditions do not occur in bulk operations mode (-b database server option).

Insensitive cursors

These cursors have insensitive membership, order, and values. No changes made after cursor open time are visible.

Insensitive cursors are used only for read-only cursor types.

Standards

Insensitive cursors correspond to the ISO/ANSI standard definition of insensitive cursors, and to ODBC static cursors.

Programming interfaces

Interface	Cursor type	Comment
ODBC, ADO/OLE DB	Static	If an updatable static cursor is requested, a value-sensitive cursor is used instead.
Embedded SQL	INSENSITIVE	
JDBC	INSENSITIVE	Insensitive semantics are only supported by the iAnywhere JDBC driver.
Open Client	Unsupported	

Description

Insensitive cursors always return rows that match the query's selection criteria, in the order specified by any ORDER BY clause.

The result set of an insensitive cursor is fully materialized as a work table when the cursor is opened. This has the following consequences:

- If the result set is very large, the disk space and memory requirements for managing the result set may be significant.
- No row is returned to the application before the entire result set is assembled as a work table. For complex queries, this may lead to a delay before the first row is returned to the application.
- Subsequent rows can be fetched directly from the work table, and so are returned quickly. The client library may prefetch several rows at a time, further improving performance.
- Insensitive cursors are not affected by ROLLBACK or ROLLBACK TO SAVEPOINT.

Sensitive cursors

Sensitive cursors can be used for read-only or updatable cursor types.

These cursors have sensitive membership, order, and values.

Standards

Sensitive cursors correspond to the ISO/ANSI standard definition of sensitive cursors, and to ODBC dynamic cursors.

Programming interfaces

Interface	Cursor type	Comment
ODBC, ADO/OLE DB	Dynamic	
Embedded SQL	SENSITIVE	Also supplied in response to a request for a DYNAMIC SCROLL cursor when no work table is required and the prefetch option is set to Off.
JDBC	SENSITIVE	Sensitive cursors are fully supported by the iAnywhere JDBC driver.

Description

Prefetching is disabled for sensitive cursors. All changes are visible through the cursor, including changes through the cursor and from other transactions. Higher isolation levels may hide some changes made in other transactions because of locking.

Changes to cursor membership, order, and all column values are all visible. For example, if a sensitive cursor contains a join, and one of the values of one of the underlying tables is modified, then all result rows composed from that base row show the new value. Result set membership and order may change at each fetch.

Sensitive cursors always return rows that match the query's selection criteria, and are in the order specified by any ORDER BY clause. Updates may affect the membership, order, and values of the result set.

The requirements of sensitive cursors place restrictions on the implementation of sensitive cursors:

- Rows cannot be prefetched, as changes to the prefetched rows would not be visible through the cursor. This may impact performance.
- Sensitive cursors must be implemented without any work tables being constructed, as changes to those rows stored as work tables would not be visible through the cursor.
- The no work table limitation restricts the choice of join method by the optimizer and therefore may impact performance.
- For some queries, the optimizer is unable to construct a plan that does not include a work table that would make a cursor sensitive.

  Work tables are commonly used for sorting and grouping intermediate results. A work table is not needed for sorting if the rows can be accessed through an index. It is not possible to state exactly which queries employ work tables, but the following queries do employ them:

  - UNION queries, although UNION ALL queries do not necessarily use work tables.
  - Statements with an ORDER BY clause, if there is no index on the ORDER BY column.
  - Any query that is optimized using a hash join.
  - Many queries involving DISTINCT or GROUP BY clauses.

In these cases, SQL Anywhere either returns an error to the application, or changes the cursor type to an asensitive cursor and returns a warning.

For more information about query optimization and the use of work tables, see "Query optimization and execution" [*SQL Anywhere Server - SQL Usage*].

Asensitive cursors

These cursors do not have well-defined sensitivity in their membership, order, or values. The flexibility that is allowed in the sensitivity permits asensitive cursors to be optimized for performance.

Asensitive cursors are used only for read-only cursor types.

Standards

Asensitive cursors correspond to the ISO/ANSI standard definition of asensitive cursors, and to ODBC cursors with unspecific sensitivity.

Programming interfaces

Interface	Cursor type
ODBC, ADO/OLE DB	Unspecified sensitivity
Embedded SQL	DYNAMIC SCROLL

Description

A request for an asensitive cursor places few restrictions on the methods SQL Anywhere can use to optimize the query and return rows to the application. For these reasons, asensitive cursors provide the best performance. In particular, the optimizer is free to employ any measure of materialization of intermediate results as work tables, and rows can be prefetched by the client.

SQL Anywhere makes no guarantees about the visibility of changes to base underlying rows. Some changes may be visible, others not. Membership and order may change at each fetch. In particular, updates to base rows may result in only some of the updated columns being reflected in the cursor's result.

Asensitive cursors do not guarantee to return rows that match the query's selection and order. The row membership is fixed at cursor open time, but subsequent changes to the underlying values are reflected in the results.

Asensitive cursors always return rows that matched the customer's WHERE and ORDER BY clauses at the time the cursor membership is established. If column values change after the cursor is opened, rows may be returned that no longer match WHERE and ORDER BY clauses.

Value-sensitive cursors

These cursors are insensitive with respect to their membership, and sensitive with respect to the order and values of the result set.

Value-sensitive cursors can be used for read-only or updatable cursor types.

Standards

Value-sensitive cursors do not correspond to an ISO/ANSI standard definition. They correspond to ODBC keyset-driven cursors.

Programming interfaces

Interface	Cursor type	Comment
ODBC, ADO/OLE DB	Keyset-driven	
Embedded SQL	SCROLL	
JDBC	INSENSITIVE and CONCUR_UPDATABLE	With the iAnywhere JDBC driver, a request for an updatable INSENSITIVE cursor is answered with a value-sensitive cursor.
Open Client and jConnect	Not supported	

Description

If the application fetches a row composed of a base underlying row that has changed, then the application must be presented with the updated value, and the SQL_ROW_UPDATED status must be issued to the application. If the application attempts to fetch a row that was composed of a base underlying row that was deleted, a SQL_ROW_DELETED status must be issued to the application.

Changes to primary key values remove the row from the result set (treated as a delete, followed by an insert). A special case occurs when a row in the result set is deleted (either from cursor or outside) and a new row with the same key value is inserted. This will result in the new row replacing the old row where it appeared.

There is no guarantee that rows in the result set match the query's selection or order specification. Since row membership is fixed at open time, subsequent changes that make a row not match the WHERE clause or ORDER BY do not change a row's membership nor position.

All values are sensitive to changes made through the cursor. The sensitivity of membership to changes made through the cursor is controlled by the ODBC option SQL_STATIC_SENSITIVITY. If this option is on, then inserts through the cursor add the row to the cursor. Otherwise, they are not part of the result set. Deletes through the cursor remove the row from the result set, preventing a hole returning the SQL_ROW_DELETED status.

Value-sensitive cursors use a **key set table**. When the cursor is opened, SQL Anywhere populates a work table with identifying information for each row contributing to the result set. When scrolling through the result set, the key set table is used to identify the membership of the result set, but values are obtained, if necessary, from the underlying tables.

The fixed membership property of value-sensitive cursors allows your application to remember row positions within a cursor and be assured that these positions will not change. For more information, see "Cursor sensitivity example: A deleted row" on page 40.

- If a row was updated or may have been updated since the cursor was opened, SQL Anywhere returns a SQLE_ROW_UPDATED_WARNING when the row is fetched. The warning is generated only once: fetching the same row again does not produce the warning.

  An update to any column of the row causes the warning, even if the updated column is not referenced by the cursor. For example, a cursor on Surname and GivenName would report the update even if only the Birthdate column was modified. These update warning and error conditions do not occur in bulk operations mode (-b database server option) when row locking is disabled. See "Performance aspects of bulk operations" [*SQL Anywhere Server - SQL Usage*].

  For more information, see "Row has been updated since last time read" [*Error Messages*].

- An attempt to execute a positioned update or delete on a row that has been modified since it was last fetched returns a SQLE_ROW_UPDATED_SINCE_READ error and cancels the statement. An application must FETCH the row again before the UPDATE or DELETE is permitted.

  An update to any column of the row causes the error, even if the updated column is not referenced by the cursor. The error does not occur in bulk operations mode.

  For more information, see "Row has changed since last read -- operation canceled" [*Error Messages*].

- If a row has been deleted after the cursor is opened, either through the cursor or from another transaction, a **hole** is created in the cursor. The membership of the cursor is fixed, so a row position is reserved, but the DELETE operation is reflected in the changed value of the row. If you fetch the row at this hole, you receive a `No Current Row of Cursor` error, indicating that there is no current row, and the cursor is left positioned on the hole. You can avoid holes by using sensitive cursors, as their membership changes along with the values.

  For more information, see "No current row of cursor" [*Error Messages*].

Rows cannot be prefetched for value-sensitive cursors. This requirement may impact performance in some cases.

Inserting multiple rows

When inserting multiple rows through a value-sensitive cursor, the new rows appear at the end of the result set. For more information, see "Modifying rows through a cursor" on page 34.

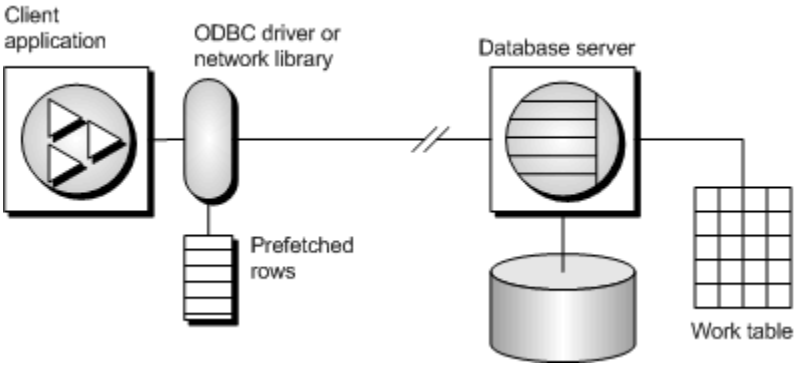
Cursor sensitivity and performance

There is a trade-off between performance and other cursor properties. In particular, making a cursor updatable places restrictions on the cursor query processing and delivery that constrain performance. Also, putting requirements on cursor sensitivity may constrain cursor performance.

To understand how the updatability and sensitivity of cursors affects performance, you need to understand how the results that are visible through a cursor are transmitted from the database to the client application.

In particular, results may be stored at two intermediate locations for performance reasons:

- **Work tables** Either intermediate or final results may be stored as work tables. Value-sensitive cursors employ a work table of primary key values. Query characteristics may also lead the optimizer to use work tables in its chosen execution plan.
- **Prefetching** The client side of the communication may retrieve rows into a buffer on the client side to avoid separate requests to the database server for each row.



Sensitivity and updatability limit the use of intermediate locations.

Prefetching rows

Prefetches and multiple-row fetches are different. Prefetches can be carried out without explicit instructions from the client application. Prefetching retrieves rows from the server into a buffer on the client side, but does not make those rows available to the client application until the application fetches the appropriate row.

By default, the SQL Anywhere client library prefetches multiple rows whenever an application fetches a single row. The SQL Anywhere client library stores the additional rows in a buffer.

Prefetching assists performance by cutting down on client/server round trips, and increases throughput by making many rows available without a separate request to the server for each row or block of rows.

For more information about controlling prefetches, see “prefetch option [database]” [*SQL Anywhere Server - Database Administration*].

Controlling prefetching from an application

- The prefetch option controls whether or not prefetching occurs. You can set the prefetch option to Always, Conditional, or Off for a single connection. By default, it is set to Conditional.
- In embedded SQL, you can control prefetching on a per-cursor basis when you open a cursor on an individual FETCH operation using the BLOCK clause.

  The application can specify a maximum number of rows contained in a single fetch from the server by specifying the BLOCK clause. For example, if you are fetching and displaying 5 rows at a time, you

could use BLOCK 5. Specifying BLOCK 0 fetches 1 record at a time and also causes a FETCH RELATIVE 0 to always fetch the row from the server again.

Although you can also turn off prefetch by setting a connection parameter on the application, it is more efficient to specify BLOCK 0 than to set the prefetch option to Off.

For more information, see "prefetch option [database]" [*SQL Anywhere Server - Database Administration*].

- Prefetch is disabled by default for value sensitive cursor types.
- In Open Client, you can control prefetching behavior using ct_cursor with CS_CURSOR_ROWS after the cursor is declared, but before it is opened.

Prefetch dynamically increases the number of prefetch rows in cases that are likely to result in improved performance. These cases include cursors for which all of the following conditions are met:

- They use one of the supported cursor types:
  - **ODBC and OLE DB** FORWARD-ONLY and READ-ONLY (default) cursors
  - **Embedded SQL** DYNAMIC SCROLL (default), NO SCROLL, and INSENSITIVE cursors
  - **ADO.NET** all cursors
- They perform only FETCH NEXT operations (no absolute, relative, or backward fetching).
- The application does not change the host variable type between fetches and does not use a GET DATA statement to get column data in chunks (using *one* GET DATA statement to get the value is supported).

Lost updates

When using an updatable cursor, it is important to guard against lost updates. A lost update is a scenario in which two or more transactions update the same row, but neither transaction is aware of the modification made by the other transaction, and consequently the second change overwrites the first modification. The following example illustrates this problem:

1. An application opens a cursor on the following query against the sample database.

```
SELECT ID, Quantity
FROM Products;
```

ID	Quantity
300	28
301	54
302	75
...	...

2. The application fetches the row with ID = 300 through the cursor.
3. A separate transaction updates the row using the following statement:


```
UPDATE Products
SET Quantity = Quantity - 10
WHERE ID = 300;
```

4. The application then updates the row through the cursor to a value of `(Quantity - 5 )`.
5. The correct final value for the row would be 13. If the cursor had prefetched the row, the new value of the row would be 23. The update from the separate transaction is lost.

In a database application, the potential for a lost update exists at any isolation level if changes are made to rows without verification of their values beforehand. At higher isolation levels (2 and 3), locking (read, intent, and write locks) can be used to ensure that changes to rows cannot be made by another transaction once the row has been read by the application. However, at isolation levels 0 and 1, the potential for lost updates is greater: at isolation level 0, read locks are not acquired to prevent subsequent changes to the data, and isolation level 1 only locks the current row. Lost updates cannot occur when using snapshot isolation since any attempt to change an old value results in an update conflict. Moreover, the use of prefetching at isolation level 1 can also introduce the potential for lost updates, since the result set row that the application is positioned on, which is in the client's prefetch buffer, may not be the same as the current row that the server is positioned on in the cursor.

To prevent lost updates from occurring with cursors at isolation level 1, the database server supports three different concurrency control mechanisms that can be specified by an application:

1. The acquisition of intent row locks on each row in the cursor as it is fetched. Intent locks prevent other transactions from acquiring intent or write locks on the same row, preventing simultaneous updates. However, intent locks do not block read row locks, so they do not affect the concurrency of read-only statements
2. The use of a value-sensitive cursor. Value-sensitive cursors can be used to track when an underlying row has changed, or has been deleted, so that the application can respond accordingly.
3. The use of FETCH FOR UPDATE, which acquires an intent row lock for that specific row.

How these alternatives are specified depends on the interface used by the application. For the first two alternatives that pertain to a SELECT statement:

- In ODBC, lost updates cannot occur because the application must specify a cursor concurrency parameter to the SQLSetStmtAttr function when declaring an updatable cursor. This parameter is one of SQL_CONCUR_LOCK, SQL_CONCUR_VALUES, SQL CONCUR_READ_ONLY, or SQL_CONCUR_TIMESTAMP. For SQL_CONCUR_LOCK, the database server acquires row intent locks. For SQL_CONCUR_VALUES and SQL_CONCUR_TIMESTAMP, a value-sensitive cursor is used. SQL_CONCUR_READ_ONLY is used for read-only cursors, and is the default.
- In JDBC, the concurrency setting for a statement is similar to that of ODBC. The iAnywhere JDBC driver supports the JDBC concurrency values RESULTSET_CONCUR_READ_ONLY and RESULTSET_CONCUR_UPDATABLE. The first value corresponds to the ODBC concurrency setting SQL_CONCUR_READ_ONLY and specifies a read-only statement. The second value corresponds to the ODBC SQL_CONCUR_LOCK setting, so row intent locks are used to prevent lost updates. Note that value-sensitive cursors cannot be specified directly in the JDBC 3.0 specification.
- In jConnect, updatable cursors are supported at the API level, but the underlying implementation (using TDS) does not support updates through a cursor. Instead, jConnect sends a separate UPDATE statement to the database server to update the specific row. To avoid lost updates, the application must run at

isolation level 2 or higher. Alternatively, the application can issue separate UPDATE statements from the cursor, but you must ensure that the UPDATE statement verifies that the row values have not been altered since the row was read by placing appropriate conditions in the UPDATE statement's WHERE clause.

- In embedded SQL, a concurrency specification can be set by including syntax within the SELECT statement itself, or in the cursor declaration. In the SELECT statement, the syntax SELECT ... FOR UPDATE BY LOCK causes the database server to acquire intent row locks on the result set.

  Alternatively, SELECT ... FOR UPDATE BY [VALUES | TIMESTAMP] causes the database server to change the cursor type to a value-sensitive cursor, so that if a specific row has been changed since the row was last read through the cursor, the application receives either a warning (SQLE_ROW_UPDATED_WARNING) on a FETCH statement, or an error (SQLE_ROW_UPDATED_SINCE_READ) on an UPDATE WHERE CURRENT OF statement. If the row was deleted, the application also receives an error (SQLE_NO_CURRENT_ROW).

FETCH FOR UPDATE functionality is also supported by the embedded SQL and ODBC interfaces, although the details differ depending on the API that is used.

In embedded SQL, the application uses FETCH FOR UPDATE, rather than FETCH, to cause an intent lock to be acquired on the row. In ODBC, the application uses the API call SQLSetPos with the operation argument SQL_POSITION or SQL_REFRESH, and the lock type argument SQL_LOCK_EXCLUSIVE, to acquire an intent lock on a row. In SQL Anywhere, these are long-term locks that are held until the transaction commits or rolls back.

Cursor sensitivity and isolation levels

Both cursor sensitivity and isolation levels address the problem of concurrency control, but in different ways, and with different sets of tradeoffs.

By choosing an isolation level for a transaction (typically at the connection level), you determine the type and locks to place, and when, on rows in the database. Locks prevent other transactions from accessing or modifying rows in the database. In general, the greater the number of locks held, the lower the expected level of concurrency across concurrent transactions.

However, locks do not prevent updates from other portions of the same transaction from occurring. Consequently, a single transaction that maintains multiple updatable cursors cannot rely on locking to prevent such problems as lost updates.

Snapshot isolation is intended to eliminate the need for read locks by ensuring that each transaction sees a consistent view of the database. The obvious advantage is that a consistent view of the database can be queried without relying on fully serializable transactions (isolation level 3), and the loss of concurrency that comes with using isolation level 3. However, snapshot isolation comes with a significant cost because copies of modified rows must be maintained to satisfy the requirements of both concurrent snapshot transactions already executing, and snapshot transactions that have yet to start. Because of this copy maintenance, the use of snapshot isolation may be inappropriate for heavy-update workloads. See “Choosing a snapshot isolation level” [*SQL Anywhere Server - SQL Usage*].

Cursor sensitivity, on the other hand, determines which changes are visible (or not) to the cursor's result. Because cursor sensitivity is specified on a cursor basis, cursor sensitivity applies to both the effects of other transactions and to update activity of the same transaction, although these effects depend entirely on the cursor type specified. By setting cursor sensitivity, you are not directly determining when locks are placed on rows in the database. However, it is the combination of cursor sensitivity and isolation level that controls the various concurrency scenarios that are possible with a particular application.

Requesting SQL Anywhere cursors

When you request a cursor type from your client application, SQL Anywhere provides a cursor. SQL Anywhere cursors are defined, not by the type as specified in the programming interface, but by the sensitivity of the result set to changes in the underlying data. Depending on the cursor type you ask for, SQL Anywhere provides a cursor with behavior to match the type.

SQL Anywhere cursor sensitivity is set in response to the client cursor type request.

ADO.NET

Forward-only, read-only cursors are available by using SACommand.ExecuteReader. The SADataAdapter object uses a client-side result set instead of cursors.

For more information, see "SACommand class" on page 192.

ADO/OLE DB and ODBC

The following table illustrates the cursor sensitivity that is set in response to different ODBC scrollable cursor types.

ODBC scrollable cursor type	SQL Anywhere cursor
STATIC	Insensitive
KEYSET-DRIVEN	Value-sensitive
DYNAMIC	Sensitive
MIXED	Value-sensitive

A MIXED cursor is obtained by setting the cursor type to SQL_CURSOR_KEYSET_DRIVEN, and then specifying the number of rows in the keyset for a keyset-driven cursor using SQL_ATTR_KEYSET_SIZE . If the keyset size is 0 (the default), the cursor is fully keyset-driven. If the keyset size is greater than 0, the cursor is mixed (keyset-driven within the keyset and dynamic outside of the keyset). The default keyset size is 0. It is an error if the keyset size is greater than 0 and less than the rowset size (SQL_ATTR_ROW_ARRAY_SIZE).

For information about SQL Anywhere cursors and their behavior, see "SQL Anywhere cursors" on page 39. For information about how to request a cursor type in ODBC, see "Choosing ODBC cursor characteristics" on page 463.

Exceptions

If a STATIC cursor is requested as updatable, a value-sensitive cursor is supplied instead and a warning is issued.

If a DYNAMIC or MIXED cursor is requested and the query cannot be executed without using work tables, a warning is issued and an asensitive cursor is supplied instead.

JDBC

The JDBC 2.0 and 3.0 specifications support three types of cursors: insensitive, sensitive, and forward-only asensitive. The iAnywhere JDBC driver is compliant with these JDBC specifications and supports these different cursor types for a JDBC ResultSet object. However, there are cases when the database server cannot construct an access plan with the required semantics for a given cursor type. In these cases, the database server either returns an error or substitutes a different cursor type. See "Sensitive cursors" on page 44.

With jConnect, the underlying protocol (TDS) only supports forward-only, read-only asensitive cursors on the database server, even though jConnect supports the APIs for creating different types of cursors following the JDBC 2.0 specification. All jConnect cursors are asensitive because the TDS protocol buffers the statement's result set in blocks. These blocks of buffered results are scrolled when the application needs to scroll through an insensitive or sensitive cursor type that supports scrollability. If the application scrolls backward past the beginning of the cached result set, the statement is re-executed, which can result in data inconsistencies if the data has been altered between statement executions.

Embedded SQL

To request a cursor from an embedded SQL application, you specify the cursor type on the DECLARE statement. The following table illustrates the cursor sensitivity that is set in response to different requests:

Cursor type	SQL Anywhere cursor
NO SCROLL	Asensitive
DYNAMIC SCROLL	Asensitive
SCROLL	Value-sensitive
INSENSITIVE	Insensitive
SENSITIVE	Sensitive

Exceptions

If a DYNAMIC SCROLL or NO SCROLL cursor is requested as UPDATABLE, then a sensitive or value-sensitive cursor is supplied. It is not guaranteed which of the two is supplied. This uncertainty fits the definition of asensitive behavior.

If an INSENSITIVE cursor is requested as UPDATABLE, then a value-sensitive cursor is supplied.

If a DYNAMIC SCROLL cursor is requested, if the prefetch database option is set to Off, and if the query execution plan involves no work tables, then a sensitive cursor may be supplied. Again, this uncertainty fits the definition of asensitive behavior.

Open Client

As with jConnect, the underlying protocol (TDS) for Open Client only supports forward-only, read-only, asensitive cursors.

Describing result sets

Some applications build SQL statements that cannot be completely specified in the application. In some cases, for example, statements depend on a response from the user before the application knows exactly what information to retrieve, such as when a reporting application allows a user to select which columns to display.

In such a case, the application needs a method for retrieving information about both the nature of the **result set** and the contents of the result set. The information about the nature of the result set, called a **descriptor**, identifies the data structure, including the number and type of columns expected to be returned. Once the application has determined the nature of the result set, retrieving the contents is straightforward.

This **result set metadata** (information about the nature and content of the data) is manipulated using descriptors. Obtaining and managing the result set metadata is called **describing**.

Since cursors generally produce result sets, descriptors and cursors are closely linked, although some interfaces hide the use of descriptors from the user. Typically, statements needing descriptors are either SELECT statements or stored procedures that return result sets.

A sequence for using a descriptor with a cursor-based operation is as follows:

1. Allocate the descriptor. This may be done implicitly, although some interfaces allow explicit allocation as well.
2. Prepare the statement.
3. Describe the statement. If the statement is a stored procedure call or batch, and the result set is not defined by a result clause in the procedure definition, then the describe should occur after opening the cursor.
4. Declare and open a cursor for the statement (embedded SQL) or execute the statement.
5. Get the descriptor and modify the allocated area if necessary. This is often done implicitly.
6. Fetch and process the statement results.
7. Deallocate the descriptor.
8. Close the cursor.
9. Drop the statement. Some interfaces do this automatically.

Implementation notes

- In embedded SQL, a SQLDA (SQL Descriptor Area) structure holds the descriptor information.

  For more information, see "The SQL descriptor area (SQLDA)" on page 541.
- In ODBC, a descriptor handle allocated using SQLAllocHandle provides access to the fields of a descriptor. You can manipulate these fields using SQLSetDescRec, SQLSetDescField, SQLGetDescRec, and SQLGetDescField.

  Alternatively, you can use SQLDescribeCol and SQLColAttributes to obtain column information.
- In Open Client, you can use ct_dynamic to prepare a statement and ct_describe to describe the result set of the statement. However, you can also use ct_command to send a SQL statement without preparing it

first and use ct_results to handle the returned rows one by one. This is the more common way of operating in Open Client application development.

- In JDBC, the java.sql.ResultSetMetaData class provides information about result sets.
- You can also use descriptors for sending data to the database server (for example, with the INSERT statement); however, this is a different kind of descriptor than for result sets.

  For more information about input and output parameters of the DESCRIBE statement, see "DESCRIBE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

Controlling transactions in applications

Transactions are sets of atomic SQL statements. Either all statements in the transaction are executed, or none. This section describes a few aspects of transactions in applications.

For more information about transactions, see "Using transactions and isolation levels" [*SQL Anywhere Server - SQL Usage*].

Setting autocommit or manual commit mode

Database programming interfaces can operate in either **manual commit** mode or **autocommit** mode.

- **Manual commit mode** Operations are committed only when your application carries out an explicit commit operation or when the database server carries out an automatic commit, for example when executing an ALTER TABLE statement or other data definition statement. Manual commit mode is also sometimes called **chained mode**.

  To use transactions in your application, including nested transactions and savepoints, you must operate in manual commit mode.

- **Autocommit mode** Each statement is treated as a separate transaction. Autocommit mode is equivalent to appending a COMMIT statement to the end of each of your SQL statements. Autocommit mode is also sometimes called **unchained mode**.

Autocommit mode can affect the performance and behavior of your application. Do not use autocommit if your application requires transactional integrity.

For information about autocommit impact on performance, see "Turn off autocommit mode" [*SQL Anywhere Server - SQL Usage*].

Controlling autocommit behavior

The way to control the commit behavior of your application depends on the programming interface you are using. The implementation of autocommit may be client-side or server-side, depending on the interface.

For more information, see "Autocommit implementation details" on page 59.

To control autocommit mode (ADO.NET)

- By default, the ADO.NET provider operates in autocommit mode. To use explicit transactions, use the SAConnection.BeginTransaction method.

  For more information, see "Transaction processing" on page 134.

To control autocommit mode (OLE DB)

- By default, the OLE DB provider operates in autocommit mode. To use explicit transactions, use the ITransactionLocal::StartTransaction, ITransaction::Commit, and ITransaction::Abort methods.

To control autocommit mode (ODBC)

- By default, ODBC operates in autocommit mode. The way you turn off autocommit depends on whether you are using ODBC directly, or using an application development tool. If you are programming directly to the ODBC interface, set the SQL_ATTR_AUTOCOMMIT connection attribute.

To control autocommit mode (JDBC)

- By default, JDBC operates in autocommit mode. To turn off autocommit, use the setAutoCommit method of the connection object:

```
conn.setAutoCommit( false );
```

To control autocommit mode (embedded SQL)

- By default, embedded SQL applications operate in manual commit mode. To turn on autocommit, set the chained database option (a server-side option) to Off using a statement such as the following:

```
SET OPTION chained='Off';
```

To control autocommit mode (Open Client)

- By default, a connection made through Open Client operates in autocommit mode. You can change this behavior by setting the chained database option (a server-side option) to On in your application using a statement such as the following:

```
SET OPTION chained='On';
```

To control autocommit mode (PHP)

- By default, PHP operates in autocommit mode. To turn off autocommit, use the sqlanywhere_set_option function:

```
$result = sqlanywhere_set_option( $conn, "auto_commit", "Off" );
```

To control autocommit mode (on the server)

- By default, the database server operates in manual commit mode. To turn on automatic commits, set the chained database option (a server-side option) to Off using a statement such as the following:

```
SET OPTION chained='Off';
```

  If you are using an interface that controls commits on the client side, setting the chained database option (a server-side option) can impact performance and/or behavior of your application. Setting the server's chained mode is not recommended.

Autocommit implementation details

Autocommit mode has slightly different behavior depending on the interface you are using and how you control the autocommit behavior.

Autocommit mode can be implemented in one of two ways:

- **Client-side autocommit** When an application uses autocommit, the client-library sends a COMMIT statement after each SQL statement executed.

ADO.NET, ADO/OLE DB, ODBC, and PHP applications control commit behavior from the client side.

- **Server-side autocommit** When an application turns off chained mode, the database server commits the results of each SQL statement. This behavior is controlled, implicitly in the case of JDBC, by the chained database option.

  Embedded SQL, JDBC, and Open Client applications manipulate server-side commit behavior (for example, they set the chained option).

There is a difference between client-side and server-side autocommit in the case of compound statements such as stored procedures or triggers. From the client side, a stored procedure is a single statement, and so autocommit sends a single commit statement after the whole procedure is executed. From the database server perspective, the stored procedure may be composed of many SQL statements, and so server-side autocommit commits the results of each SQL statement within the procedure.

Do not mix client-side and server-side implementations
Do not combine setting of the chained option with setting of the autocommit option in your ADO.NET, ADO/OLE DB, ODBC, or PHP application.

Controlling the isolation level

You can set the isolation level of a current connection using the isolation_level database option.

Some interfaces, such as ODBC, allow you to set the isolation level for a connection at connection time. You can reset this level later using the isolation_level database option. See “isolation_level option [compatibility]” [*SQL Anywhere Server - Database Administration*].

Cursors and transactions

In general, a cursor closes when a COMMIT is performed. There are two exceptions to this behavior:

- The close_on_endtrans database option is set to Off.
- A cursor is opened WITH HOLD, which is the default with Open Client and JDBC.

If either of these two cases is true, the cursor remains open on a COMMIT.

ROLLBACK and cursors

If a transaction rolls back, then cursors close except for those cursors opened WITH HOLD. However, don't rely on the contents of any cursor after a rollback.

The draft ISO SQL3 standard states that on a rollback, all cursors (even those cursors opened WITH HOLD) should close. You can obtain this behavior by setting the ansi_close_cursors_on_rollback option to On.

Savepoints

If a transaction rolls back to a savepoint, and if the ansi_close_cursors_on_rollback option is On, then all cursors (even those cursors opened WITH HOLD) opened after the SAVEPOINT close.

Cursors and isolation levels

You can change the isolation level of a connection during a transaction using the SET OPTION statement to alter the isolation_level option. However, this change does not affect open cursors.

CHAPTER 4

Three-tier computing and distributed transactions

Contents

Introduction to three-tier computing and distributed transactions

You can use SQL Anywhere as a database server or **resource manager**, participating in distributed transactions coordinated by a transaction server.

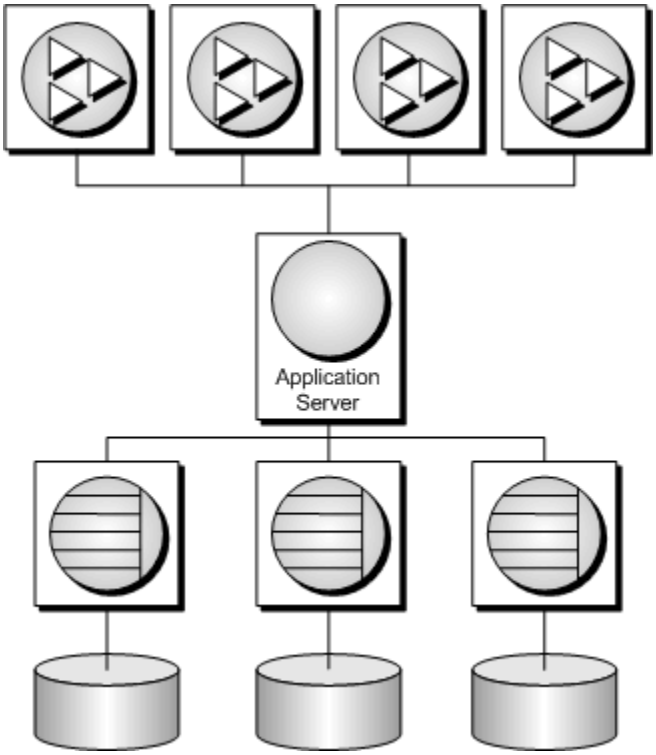
A three-tier environment, where an application server sits between client applications and a set of resource managers, is a common distributed-transaction environment. Sybase EAServer and some other application servers are also transaction servers.

Sybase EAServer and Microsoft Transaction Server both use the Microsoft Distributed Transaction Coordinator (DTC) to coordinate transactions. SQL Anywhere provides support for distributed transactions controlled by the DTC service, so you can use SQL Anywhere with either of these application servers, or any other product based on the DTC model.

When integrating SQL Anywhere into a three-tier environment, most of the work needs to be done from the Application Server. This chapter provides an introduction to the concepts and architecture of three-tier computing, and an overview of relevant SQL Anywhere features. It does not describe how to configure your Application Server to work with SQL Anywhere. For more information, see your Application Server documentation.

Three-tier computing architecture

In three-tier computing, application logic is held in an application server, such as Sybase EAServer, which sits between the resource manager and the client applications. In many situations, a single application server may access multiple resource managers. In the Internet case, client applications are browser-based, and the application server is generally a web server extension.



Sybase EAServer stores application logic in the form of components, and makes these components available to client applications. The components may be PowerBuilder components, JavaBeans, or COM components.

For more information, see your Sybase EAServer documentation.

Distributed transactions in three-tier computing

When client applications or application servers work with a single transaction processing database, such as SQL Anywhere, there is no need for transaction logic outside the database itself, but when working with multiple resource managers, transaction control must span the resources involved in the transaction. Application servers provide transaction logic to their client applications—guaranteeing that sets of operations are executed atomically.

Many transaction servers, including Sybase EAServer, use the Microsoft Distributed Transaction Coordinator (DTC) to provide transaction services to their client applications. DTC uses **OLE**

transactions, which in turn use the **two-phase commit** protocol to coordinate transactions involving multiple resource managers. You must have DTC installed to use the features described in this chapter.

SQL Anywhere in distributed transactions

SQL Anywhere can take part in transactions coordinated by DTC, which means that you can use SQL Anywhere databases in distributed transactions using a transaction server such as Sybase EAServer or Microsoft Transaction Server. You can also use DTC directly in your applications to coordinate transactions across multiple resource managers.

The vocabulary of distributed transactions

This chapter assumes some familiarity with distributed transactions. For information, see your transaction server documentation. This section describes some commonly used terms.

- **Resource managers** are those services that manage the data involved in the transaction.

  The SQL Anywhere database server can act as a resource manager in a distributed transaction when accessed through OLE DB or ODBC. The ODBC driver and OLE DB provider act as resource manager proxies on the client computer.

- Instead of communicating directly with the resource manager, application components can communicate with **resource dispensers**, which in turn manage connections or pools of connections to the resource managers.

  SQL Anywhere supports two resource dispensers: the ODBC driver manager and OLE DB.

- When a transactional component requests a database connection (using a resource manager), the application server **enlists** each database connection that takes part in the transaction. DTC and the resource dispenser perform the enlistment process.

Two-phase commit

Distributed transactions are managed using two-phase commit. When the work of the transaction is complete, the transaction manager (DTC) asks all the resource managers enlisted in the transaction whether they are ready to commit the transaction. This phase is called **preparing** to commit.

If all the resource managers respond that they are prepared to commit, DTC sends a commit request to each resource manager, and responds to its client that the transaction is completed. If one or more resource manager does not respond, or responds that it cannot commit the transaction, all the work of the transaction is rolled back across all resource managers.

How application servers use DTC

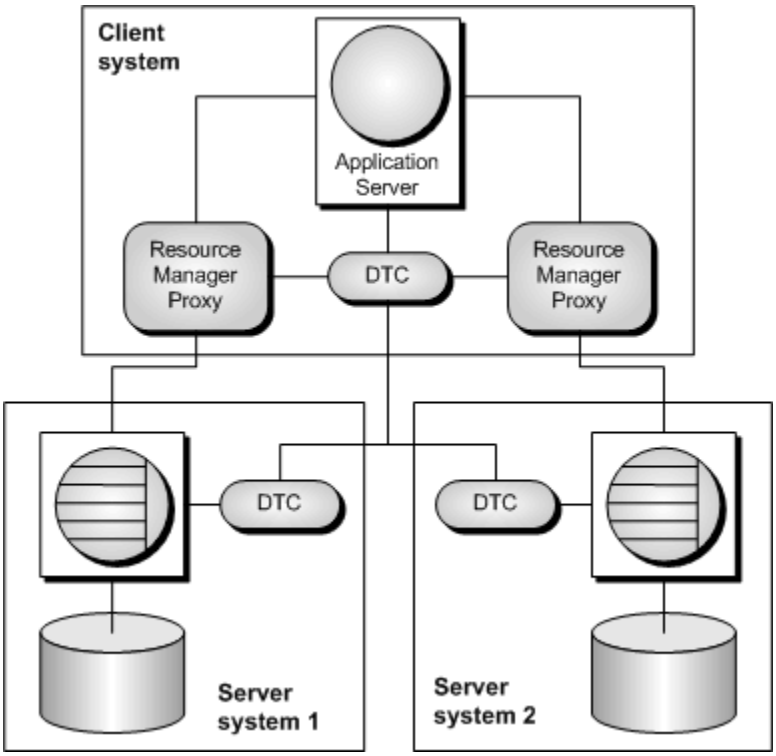
Sybase EAServer and Microsoft Transaction Server are both component servers. The application logic is held in the form of components, and made available to client applications.

Each component has a transaction attribute that indicates how the component participates in transactions. The application developer building the component must program the work of the transaction into the

component—the resource manager connections, the operations on the data for which each resource manager is responsible. However, the application developer does not need to add transaction management logic to the component. Once the transaction attribute is set, to indicate that the component needs transaction management, EAServer uses DTC to enlist the transaction and manage the two-phase commit process.

Distributed transaction architecture

The following diagram illustrates the architecture of distributed transactions. In this case, the resource manager proxy is either ODBC or OLE DB.



In this case, a single resource dispenser is used. The Application Server asks DTC to prepare a transaction. DTC and the resource dispenser enlist each connection in the transaction. Each resource manager must be in contact with both DTC and the database, so as to perform the work and to notify DTC of its transaction status when required.

A DTC service must be running on each computer to operate distributed transactions. You can control DTC services from the Services icon in the Windows Control Panel; the DTC service is named **MSDTC**.

For more information, see your DTC or EAServer documentation.

Using distributed transactions

While SQL Anywhere is enlisted in a distributed transaction, it hands transaction control over to the transaction server, and SQL Anywhere ensures that it does not perform any implicit transaction management. The following conditions are imposed automatically by SQL Anywhere when it participates in distributed transactions:

- Autocommit is automatically turned off, if it is in use.
- Data definition statements (which commit as a side effect) are disallowed during distributed transactions.
- An explicit COMMIT or ROLLBACK issued by the application directly to SQL Anywhere, instead of through the transaction coordinator, generates an error. The transaction is not aborted, however.
- A connection can participate in only a single distributed transaction at a time.
- There must be no uncommitted operations at the time the connection is enlisted in a distributed transaction.

DTC isolation levels

DTC has a set of isolation levels, which the application server specifies. These isolation levels map to SQL Anywhere isolation levels as follows:

DTC isolation level	SQL Anywhere isolation level
ISOLATIONLEVEL_UNSPECIFIED	0
ISOLATIONLEVEL_CHAOS	0
ISOLATIONLEVEL_READUNCOMMITTED	0
ISOLATIONLEVEL_BROWSE	0
ISOLATIONLEVEL_CURSORSTABILITY	1
ISOLATIONLEVEL_READCOMMITTED	1
ISOLATIONLEVEL_REPEATABLEREAD	2
ISOLATIONLEVEL_SERIALIZABLE	3
ISOLATIONLEVEL_ISOLATED	3

Recovery from distributed transactions

If the database server faults while uncommitted operations are pending, it must either rollback or commit those operations on startup to preserve the atomic nature of the transaction.

If uncommitted operations from a distributed transaction are found during recovery, the database server attempts to connect to DTC and requests that it be re-enlisted in the pending or in-doubt transactions. Once the re-enlistment is complete, DTC instructs the database server to roll back or commit the outstanding operations.

If the reenlistment process fails, SQL Anywhere has no way of knowing whether the in-doubt operations should be committed or rolled back, and recovery fails. If you want the database in such a state to recover, regardless of the uncertain state of the data, you can force recovery using the following database server options:

- **-tmf** If DTC cannot be located, the outstanding operations are rolled back and recovery continues. See "-tmf server option" [*SQL Anywhere Server - Database Administration*].
- **-tmt** If re-enlistment is not achieved before the specified time, the outstanding operations are rolled back and recovery continues. See "-tmt server option" [*SQL Anywhere Server - Database Administration*].

Using EAServer with SQL Anywhere

This section provides an overview of the actions you need to take in EAServer 3.0 or later to work with SQL Anywhere. For more detailed information, see the EAServer documentation.

Configuring EAServer

All components installed in a Sybase EAServer system share the same transaction coordinator.

EAServer 3.0 and later offer a choice of transaction coordinators. You must use DTC as the transaction coordinator if you are including SQL Anywhere in the transactions. This section describes how to configure EAServer 3.0 to use DTC as its transaction coordinator.

The component server in EAServer is named Jaguar.

To configure an EAServer to use the Microsoft DTC transaction model

1. Ensure that your Jaguar server is running.

   On Windows, the Jaguar server commonly runs as a service. To manually start the installed Jaguar server that is included with EAServer 3.0, choose **Start** » **Programs** » **Sybase** » **EAServer** » **EAServer**.

2. Start Jaguar Manager.

   From the Windows desktop, choose **Start** » **Programs** » **Sybase** » **EAServer** » **Jaguar Manager**.

3. Connect to the Jaguar server from Jaguar Manager.

   From Sybase Central, choose **Tools** » **Connect** » **Jaguar Manager**. In the connection window, enter **jagadmin** as the **User Name**, leave the **Password** field blank, and enter a **Host Name** of **localhost**. Click **OK** to connect.

4. Set the transaction model for the Jaguar server.

   In the left pane, open the **Servers** folder. In the right pane, right-click the server you want to configure, and choose **Server Properties** from the popup menu. Click the **Transactions** tab, and choose **Microsoft DTC** as the transaction model. Click **OK** to complete the operation.

Setting the component transaction attribute

In EAServer, you can implement a component that carries out operations on more than one database. You assign a **transaction attribute** to this component that defines how it participates in transactions. The transaction attribute can have the following values:

- **Not Supported** The component's methods never execute as part of a transaction. If the component is activated by another component that is executing within a transaction, the new instance's work is performed outside the existing transaction. This is the default.
- **Supports Transaction** The component can execute in the context of a transaction, but a connection is not required to execute the component's methods. If the component is instantiated directly by a base

client, EAServer does not begin a transaction. If component A is instantiated by component B, and component B is executing within a transaction, component A executes in the same transaction.

- **Requires Transaction** The component always executes in a transaction. When the component is instantiated directly by a base client, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A executes within the same transaction; if B is not executing in a transaction, then A executes in a new transaction.
- **Requires New Transaction** Whenever the component is instantiated, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A begins a new transaction that is unaffected by the outcome of B's transaction; if B is not executing in a transaction, then A executes in a new transaction.

For example, in the Sybase Virtual University sample application, included with EAServer as the SVU package, the SVUEnrollment component enroll method carries out two separate operations (reserves a seat in a course, bills the student for the course). These two operations need to be treated as a single transaction.

Microsoft Transaction Server provides the same set of attribute values.

To set the transaction attribute of a component

1. In Jaguar Manager, locate the component.

   To find the SVUEnrollment component in the Jaguar sample application, connect to the Jaguar server, open the **Packages** folder, and open the SVU package. The components in the package are listed in the right pane.

2. Set the transaction attribute for the component.

   Right-click the component, and choose **Component Properties** from the popup menu. Click the **Transaction** tab, and choose the transaction attribute value from the list. Click **OK** to complete the operation.

   The SVUEnrollment component is already marked as **Requires Transaction**.

Once the component transaction attribute is set, you can perform SQL Anywhere database operations from that component, and be assured of transaction processing at the level you have specified.

Part II. Java in the database

This part provides an introduction to Java and Java in the database.

CHAPTER 5

Java support in SQL Anywhere

Contents

Introduction to Java support

SQL Anywhere provides a mechanism for executing Java classes from within the database server environment. Using Java methods in the database server provides powerful ways of adding programming logic to a database.

Java support in the database offers the following:

- You can reuse Java components in the different layers of your application—client, middle-tier, or server—and use them wherever it makes the most sense to you. SQL Anywhere becomes a platform for distributed computing.
- Java provides a more powerful language than the SQL stored procedure language for building logic into the database.
- Java can be used in the database server without jeopardizing the integrity, security, or robustness of the database and the server.

The SQLJ standard

Java in the database is based on the SQLJ Part 1 proposed standard (ANSI/INCITS 331.1-1999). SQLJ Part 1 provides specifications for calling Java static methods as SQL stored procedures and functions.

Learning about Java in the database

The following table outlines the documentation regarding the use of Java in the database.

Title	Purpose
"Java support in SQL Anywhere" on page 75 (this chapter)	Java concepts and how to apply them in SQL Anywhere.
"Tutorial: Using Java in the database" on page 87	Practical steps for using Java in the database.
"SQL Anywhere JDBC API" on page 477	Accessing data from Java classes, including distributed computing.

The following table is a guide to which parts of the Java documentation apply to you, depending on your interests and background.

If you ...	Consider reading ...
Are a Java developer who wants to just get started.	"The runtime environment for Java in the database" on page 83 "Introduction to Java in the database tutorial" on page 88

If you ...	Consider reading ...
Want to know the key features of Java in the database.	“Java in the database Q & A” on page 78
Want to find out how to access data from Java.	“SQL Anywhere JDBC API” on page 477

Java in the database Q & A

This section describes the key features of Java in the database.

What are the key features of Java in the database?

Detailed explanations of all the following points appear in later sections.

- **You can run Java in the database server** An external Java Virtual Machine (VM) runs Java code in the database server.
- **You can access data from Java** An internal JDBC driver lets you access data from Java.
- **SQL is preserved** The use of Java does not alter the behavior of existing SQL statements or other aspects of non-Java relational database behavior.

How do I store Java classes in the database?

Java is an object-oriented language, so its instructions (source code) come in the form of classes. To execute Java in a database, you write the Java instructions outside the database and compile them outside the database into compiled classes (**byte code**), which are binary files holding Java instructions.

You then install these compiled classes into a database. Once installed, you can execute these classes in the database server as a stored procedure. For example, the following statement creates a Java stored procedure:

```
CREATE PROCEDURE insertfix()
EXTERNAL NAME 'JDBCExample.InsertFixed ()V'
LANGUAGE JAVA;
```

SQL Anywhere is a runtime environment for Java classes, not a Java development environment. You need a Java development environment, such as the Sun Microsystems Java Development Kit, to write and compile Java.

For more information, see “Installing Java classes into a database” on page 95.

How does Java get executed in a database?

SQL Anywhere uses a **Java Virtual Machine** (**VM**). The Java VM interprets compiled Java instructions and runs them in the database server. The database server starts the Java VM automatically when needed: you do not have to take any explicit action to start or stop the Java VM.

The SQL request processor in the database server has been extended so it can call into the Java VM to execute Java instructions. It can also process requests from the Java VM to enable data access from Java.

Why Java?

Java provides a number of features that make it ideal for use in the database:

- Thorough error checking at compile time.
- Built-in error handing with a well-defined error handling methodology.
- Built-in garbage collection (memory recovery).
- Elimination of many bug-prone programming techniques.
- Strong security features.
- Java code is interpreted, so no operations get executed without being acceptable to the Java VM.

On what platforms is Java in the database supported?

Java in the database is supported on all Unix and Windows operating systems except Windows Mobile.

How do I use Java and SQL together?

Java methods are declared as stored procedures, and can then be called just like SQL stored procedures.

You can use many of the classes that are part of the Java API as included in the Sun Microsystems Java Development Kit. You can also use classes created and compiled by Java developers.

How do I access Java from SQL?

You can treat Java methods as stored procedures, which can be called from SQL.

You must create a stored procedure that runs your method. For example:

```
CREATE PROCEDURE javaproc()
EXTERNAL NAME 'JDBCExample.MyMethod ()V'
LANGUAGE JAVA;
```

For more information, see “CREATE PROCEDURE statement” [*SQL Anywhere Server - SQL Reference*].

For example, the SQL function PI(*) returns the value for pi. The Java API class java.lang.Math has a parallel field named PI returning the same value. But java.lang.Math also has a field named E that returns the base of the natural logarithms, as well as a method that computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.

Other members of the Java API offer even more specialized functionality. For example, java.util.Stack generates a last-in, first-out queue that can store ordered lists; java.util.HashTable maps values to keys; and java.util.StringTokenizer breaks a string of characters into individual word units.

How can I use my own Java classes in databases?

You can install your own Java classes into a database. For example, you could design, write in Java, and compile with a Java compiler, a user-created Employees class or Package class.

User-created Java classes can contain both information about the subject and some computational logic. Once installed in a database, SQL Anywhere lets you use these classes in all parts and operations of the database and execute their functionality (in the form of class or instance methods) as easily as calling a stored procedure.

Java classes and stored procedures are different
Java classes are different from stored procedures. Whereas stored procedures are written in SQL, Java classes provide a more powerful language, and can be called from client applications as easily and in the same way as stored procedures.

For more information, see "Installing Java classes into a database" on page 95.

Can I access data using Java?

The JDBC interface is an industry standard, designed specifically to access database systems. The JDBC classes are designed to connect to a database, request data using SQL statements, and return result sets that can be processed in the client application.

Normally, client applications use JDBC classes, and the database system vendor supplies a JDBC driver that allows the JDBC classes to establish a connection.

You can connect to SQL Anywhere from a client application via JDBC, using jConnect, or using the iAnywhere JDBC driver. SQL Anywhere also provides an internal JDBC driver that permits Java classes installed in a database to use JDBC classes that execute SQL statements. See "SQL Anywhere JDBC API" on page 477.

Can I move classes from client to server?

You can create Java classes that can be moved between levels of an enterprise application. The same Java class can be integrated into either the client application, a middle tier, or the database—wherever is most appropriate.

You can move a class containing business logic to any level of the enterprise system, including the database server, allowing you complete flexibility to make the most appropriate use of resources. It also enables enterprise customers to develop their applications using a single programming language in a multi-tier architecture with unparalleled flexibility.

What can I not do with Java in the database?

SQL Anywhere is a runtime environment for Java classes, not a Java development environment.

You cannot perform the following tasks in the database:

- Edit class source files (**.java* files).
- Compile Java class source files (**.java* files).
- Execute unsupported Java APIs, such as applet and visual classes.
- Execute Java methods that require the execution of native methods. All user classes installed into the database must be 100% Java.

The Java classes used in SQL Anywhere must be written and compiled using a Java application development tool, and then installed into a database for use.

Java error handling

Java error handling code is separate from the code for normal processing.

Errors generate an exception object representing the error. This is called **throwing an exception**. A thrown exception terminates a Java program unless it is caught and handled properly at some level of the application.

Both Java API classes and custom-created classes can throw exceptions. In fact, users can create their own exception classes that throw their own custom-created classes.

If there is no exception handler in the body of the method where the exception occurred, then the search for an exception handler continues up the call stack. If the top of the call stack is reached and no exception handler has been found, the default exception handler of the Java interpreter running the application is called and the program terminates.

In SQL Anywhere, if a SQL statement calls a Java method, and an unhandled exception is thrown, a SQL error is generated.

The runtime environment for Java in the database

This section describes the SQL Anywhere runtime environment for Java, and how it differs from a standard Java runtime environment.

The runtime Java classes

The runtime Java classes are the low-level classes that are made available to a database when it is created or Java-enabled. These classes include a subset of the Java API. These classes are part of the Sun Java Development Kit.

The runtime classes provide basic functionality on which to build applications. The runtime classes are always available to classes in the database.

You can incorporate the runtime Java classes in your own user-created classes: either inheriting their functionality or using it within a calculation or operation in a method.

Examples

Some Java API classes included in the runtime Java classes include:

- **Primitive Java data types** All primitive (native) data types in Java have a corresponding class. In addition to being able to create objects of these types, the classes have additional, often useful, functionality.

  The Java int data type has a corresponding class in java.lang.Integer.

- **The utility package** The package java.util.* contains a number of very helpful classes whose functionality has no parallel in the SQL functions available in SQL Anywhere.

  Some of the classes include:

  - **Hashtable** maps keys to values.
  - **StringTokenizer** breaks a String down into individual words.
  - **Vector** holds an array of objects whose size can change dynamically.
  - **Stack** holds a last-in, first-out stack of objects.

- **JDBC for SQL operations** The package java.SQL.* contains the classes needed by Java objects to extract data from the database using SQL statements.

Unlike user-defined classes, the runtime classes are not stored in the database. Instead, they are stored in files in the *java* subdirectory of your SQL Anywhere installation directory.

Java is case sensitive

Java syntax works as you would expect it to, and SQL syntax is unaltered by the presence of Java classes. This is true even if the same SQL statement contains both Java and SQL syntax. It is a simple statement, but with far-reaching implications.

Java is case sensitive. The Java class FindOut is a completely different class from the class Findout. SQL is case insensitive with respect to keywords and identifiers.

Java case sensitivity is preserved even when embedded in a SQL statement that is case insensitive. The Java parts of the statement must be case sensitive, even though the parts previous to and following the Java syntax can be in either upper or lowercase.

For example, the following SQL statement executes successfully because the case of Java objects, classes, and operators is respected even though there is variation in the case of the remaining SQL parts of the statement.

```
SeLeCt java.lang.Math.random();
```

Strings in Java and SQL

A set of double quotes identifies string literals in Java, as in the following Java code fragment:

```
String str = "This is a string";
```

In SQL, however, single quotes mark strings, and double quotes indicate an identifier, as illustrated by the following SQL statement:

```
INSERT INTO TABLE DBA.t1
VALUES( 'Hello' );
```

You should always use the double quote in Java source code, and single quotes in SQL statements.

The following Java code fragment is valid, if used within a Java class.

```
String str = new java.lang.String(
             "Brand new object" );
```

Printing to the command line

Printing to the standard output is a quick way of checking variable values and execution results at various points of code execution. When the method in the second line of the following Java code fragment is encountered, the string argument it accepts prints out to standard output.

```
String str = "Hello world";
System.out.println( str );
```

In SQL Anywhere, standard output is the database server messages window, so the string appears there. Executing the above Java code within the database is the equivalent of the following SQL statement.

```
MESSAGE 'Hello world';
```


Using the main method

When a class contains a main method matching the following declaration, most Java run time environments, such as the Sun Java interpreter, execute it automatically. Normally, this static method executes only if it is the class being invoked by the Java interpreter.

```
public static void main( String args[ ] ) { }
```

You are always guaranteed this method will be called first when the Sun Java runtime system starts.

In SQL Anywhere, the Java runtime system is always available. The functionality of objects and methods can be tested in an ad hoc, dynamic manner using SQL statements. This provides a flexible method for testing Java class functionality.

Persistence

Once a Java class has been added to a database, it remains there until you explicitly remove it with a REMOVE JAVA statement.

Variables in Java classes, like SQL variables, persist only for the duration of the connection.

For more information about removing classes, see "REMOVE JAVA statement" [*SQL Anywhere Server - SQL Reference*].

Java escape characters in SQL statements

In Java code, you can use escape characters to insert certain special characters into strings. Consider the following code, which inserts a new line and tab in front of a sentence containing an apostrophe.

```
String str = "\n\t\This is an object\'s string literal";
```

SQL Anywhere permits the use of Java escape characters only when being used by Java classes. From within SQL, however, you must follow the rules that apply to strings in SQL.

For example, to pass a string value to a field using a SQL statement, you could use the following statement (which includes SQL escape characters), but the Java escape characters could not be used.

```
SET obj.str = '\nThis is the object''s string field';
```

For more information about SQL string handling rules, see "Strings" [*SQL Anywhere Server - SQL Reference*].

Using import statements

It is common in a Java class declaration to include an import statement to access classes in another package. You can reference imported classes using unqualified class names.

For example, you can reference the Stack class of the java.util package in two ways:

- explicitly using the name java.util.Stack
- using the name Stack, and including the following import statement:

```
import java.util.*;
```

Classes further up in the hierarchy must also be installed

A class referenced by another class, either explicitly with a fully qualified name or implicitly using an import statement, must also be installed in the database.

The import statement works as intended within compiled classes. However, within the SQL Anywhere runtime environment, no equivalent to the import statement exists. All class names used in stored procedures must be fully qualified. For example, to create a variable of type String, you would reference the class using the fully qualified name: java.lang.String.

Public fields

It is a common practice in object-oriented programming to define class fields as private and make their values available only through public methods.

Many of the examples used in this documentation render fields public to make examples more compact and easier to read. Using public fields in SQL Anywhere also offers a performance advantage over accessing public methods.

The general convention followed in this documentation is that a user-created Java class designed for use in SQL Anywhere exposes its main values in its fields. Methods contain computational automation and logic that may act on these fields.

CHAPTER 6

Tutorial: Using Java in the database

Contents

Introduction to Java in the database tutorial

This chapter describes how to accomplish tasks using Java in the database. The first thing you need to do is compile and install Java classes in a database to make them available for use in SQL Anywhere.

The following is a brief introduction to the steps involved in creating Java methods and calling them from SQL. It describes how to compile and install a Java class into the database. It also describes how to access the class and its members and methods from SQL statements.

Requirements

The tutorial assumes that you have a Java Development Kit (JDK) installed, including the Java compiler (javac) and Java VM.

Resources

Source code and batch files for this sample are provided in *samples-dir\SQLAnywhere\JavaInvoice*.

Creating and compiling the sample Java class

The first step to using Java in the database is to write the Java code and compile it. This is done outside the database.

To create and compile the class

1. Create the sample Java class source file.

   For your convenience, the sample code is included here. You can paste the following code into *Invoice.java* or obtain the file from *samples-dir\SQLAnywhere\JavaInvoice*.

```
import java.io.*;

public class Invoice
{
    public static String lineItem1Description;
    public static double lineItem1Cost;

    public static String lineItem2Description;
    public static double lineItem2Cost;

    public static double totalSum() {
        double runningsum;
        double taxfactor = 1 + Invoice.rateOfTaxation();

        runningsum = lineItem1Cost + lineItem2Cost;
        runningsum = runningsum * taxfactor;

        return runningsum;
    }

    public static double rateOfTaxation()
    {
        double rate;
        rate = .15;

        return rate;
```



```
    }

    public static void init(
      String item1desc, double item1cost,
      String item2desc, double item2cost )
    {
        lineItem1Description = item1desc;
        lineItem1Cost = item1cost;
        lineItem2Description = item2desc;
        lineItem2Cost = item2cost;
    }

    public static String getLineItem1Description()
    {
        return lineItem1Description;
    }

    public static double getLineItem1Cost()
    {
        return lineItem1Cost;
    }

    public static String getLineItem2Description()
    {
        return lineItem2Description;
    }

    public static double getLineItem2Cost()
    {
        return lineItem2Cost;
    }

    public static boolean testOut( int[] param )
    {
        param[0] = 123;
        return true;
    }

    public static void main( String[] args )
    {
        System.out.print( "Hello" );
        for ( int i = 0; i  < args.length; i++ )
            System.out.print( " " + args[i] );
        System.out.println();
    }
}
```

2. Compile the file to create the file *Invoice.class*.

   ```
   javac Invoice.java
   ```

   The class is now compiled and ready to be installed into the database.

Choosing a Java VM

The database server must be set up to locate a Java VM. Since you can specify different Java VMs for each database, the ALTER EXTERNAL ENVIRONMENT statement can be used to indicate the location (path) of the Java VM.

```
ALTER EXTERNAL ENVIRONMENT JAVA
LOCATION 'c:\\jdk1.5.0_06\\jre\\bin\\java.exe';
```

If the location is not set, the database server searches for the location of the Java VM, as follows:

- Check the JAVA_HOME environment variable.
- Check the JAVAHOME environment variable.
- Check the path.
- If the information is not in the path, return an error.

See "ALTER EXTERNAL ENVIRONMENT statement" [*SQL Anywhere Server - SQL Reference*].

Note
JAVA_HOME and JAVAHOME are environment variables commonly created when installing a Java VM. If neither of these exist, you can create them manually, and point them to the root directory of your Java VM. However, this is not required if you use the ALTER EXTERNAL ENVIRONMENT statement.

To specify the location of the Java VM (Interactive SQL)

1. Start Interactive SQL and connect to the database.
2. In the SQL Statements pane, type the following statement:

   ```
   ALTER EXTERNAL ENVIRONMENT JAVA
   LOCATION 'path\\java.exe';
   ```

   Here, *path* indicates the location of the Java VM (for example, *c:\\jdk1.5.0_06\\jre\\bin*).

You can also use the ALTER EXTERNAL ENVIRONMENT to specify the database user whose connection can be used for installing classes and performing other Java-related administrative tasks.

```
ALTER EXTERNAL ENVIRONMENT JAVA
    USER user_name
```

For more information, see "ALTER EXTERNAL ENVIRONMENT statement" [*SQL Anywhere Server - SQL Reference*].

Use the java_vm_options option to specify any additional command line options that are required to start the Java VM.

```
SET OPTION PUBLIC.java_vm_options='java-options';
```

For more information, see "java_vm_options option [database]" [*SQL Anywhere Server - Database Administration*].

If you want to use JAVA in the database, but do not have a Java Runtime Environment (JRE) installed, you can install and use any Java JRE that you want to. Once installed, it is best to set the JAVA_HOME or JAVAHOME environment variable to point to the root of the installed JRE. Note that most Java installers set one of these environment variables by default. Once a JRE is installed and JAVA_HOME or JAVAHOME is set correctly, you should then be able to use Java in the database without performing any additional steps.

Install the sample Java class

Java classes must be installed into a database before they can be used. You can install classes from Sybase Central or Interactive SQL.

To install the class to the SQL Anywhere sample database (Sybase Central)

1. Start Sybase Central and connect to the sample database.
2. Open the **External Environments** folder.
3. Under this folder, open the **Java** folder.
4. From the **File** menu, choose **New** » **Java Class**.

   The **Create Java Class Wizard** appears.
5. Use the **Browse** button to locate *Invoice.class*.
6. Click **Finish** to close the wizard.

To install the class to the SQL Anywhere sample database (Interactive SQL)

1. Start Interactive SQL and connect to the sample database.
2. In the **SQL Statements** pane of Interactive SQL, type the following statement:

   ```
   INSTALL JAVA NEW
   FROM FILE 'path\\Invoice.class';
   ```

   Here *path* is the location of your compiled class file.
3. Press F5 to execute the statement.

The class is now installed into the sample database.

Notes

- At this point, no Java in the database operations have taken place. The class has been installed into the database and is ready for use.
- Changes made to the class file from now on are *not* automatically reflected in the copy of the class in the database. You must update the classes in the database if you want the changes reflected.

For more information about installing classes, and for information about updating an installed class, see "Installing Java classes into a database" on page 95.

Using the CLASSPATH variable

The Sun Java runtime environment and the Sun JDK Java compiler use the CLASSPATH environment variable to locate classes referenced within Java code. A CLASSPATH variable provides the link between Java code and the actual file path or URL location of the classes being referenced. For example, `import java.io.*` allows all the classes in the java.io package to be referenced without a fully qualified name. Only the class name is required in the following Java code to use classes from the java.io package. The

CLASSPATH environment variable on the system where the Java class declaration is to be compiled must include the location of the Java directory, the root of the java.io package.

CLASSPATH used to install classes

The CLASSPATH variable can be used to locate a file during the installation of classes. For example, the following statement installs a user-created Java class to a database, but only specifies the name of the file, not its full path and name. (Note that this statement involves no Java operations.)

```
INSTALL JAVA NEW
FROM FILE 'Invoice.class';
```

If the file specified is in a directory or ZIP file specified by the CLASSPATH environmental variable, SQL Anywhere successfully locates the file and install the class.

Accessing methods in the Java class

To access the Java methods in the class, you must create stored procedures or functions that act as wrappers for the methods in the class.

To call a Java method using Interactive SQL

1. Create the following SQL stored procedure to call the Invoice.main method in the sample class:

   ```
   CREATE PROCEDURE InvoiceMain( IN arg1 CHAR(50) )
   EXTERNAL NAME 'Invoice.main([Ljava/lang/String;)V'
   LANGUAGE JAVA;
   ```

   This stored procedure acts as a wrapper to the Java method.

   For more information about the syntax of this statement, see "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*].

2. Call the stored procedure to call the Java method:

   ```
   CALL InvoiceMain('to you');
   ```

   If you examine the database server message log, you see the message "Hello to you" written there. The database server has redirected the output there from System.out.

Accessing fields and methods of the Java object

Here are more examples of how to call Java methods, pass arguments, and return values.

To create stored procedures/functions for the methods in the Invoice class

1. Create the following SQL stored procedures to pass arguments to and retrieve return values from the Java methods in the Invoice class:

   ```
   -- Invoice.init takes a string argument (Ljava/lang/String;)
   -- a double (D), a string argument (Ljava/lang/String;), and
   -- another double (D), and returns nothing (V)
   CREATE PROCEDURE init( IN arg1 CHAR(50),
   ```



```
                        IN arg2 DOUBLE,
                        IN arg3 CHAR(50),
                        IN arg4 DOUBLE)
EXTERNAL NAME
  'Invoice.init(Ljava/lang/String;DLjava/lang/String;D)V'
LANGUAGE JAVA;
-- Invoice.rateOfTaxation take no arguments ()
-- and returns a double (D)
CREATE FUNCTION rateOfTaxation()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.rateOfTaxation()D'
LANGUAGE JAVA;
-- Invoice.rateOfTaxation take no arguments ()
-- and returns a double (D)
CREATE FUNCTION totalSum()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.totalSum()D'
LANGUAGE JAVA;
-- Invoice.getLineItem1Description take no arguments ()
-- and returns a string (Ljava/lang/String;)
CREATE FUNCTION getLineItem1Description()
RETURNS CHAR(50)
EXTERNAL NAME
  'Invoice.getLineItem1Description()Ljava/lang/String;'
LANGUAGE JAVA;
-- Invoice.getLineItem1Cost take no arguments ()
-- and returns a double (D)
CREATE FUNCTION getLineItem1Cost()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.getLineItem1Cost()D'
LANGUAGE JAVA;
-- Invoice.getLineItem2Description take no arguments ()
-- and returns a string (Ljava/lang/String;)
CREATE FUNCTION getLineItem2Description()
RETURNS CHAR(50)
EXTERNAL NAME
  'Invoice.getLineItem2Description()Ljava/lang/String;'
LANGUAGE JAVA;
-- Invoice.getLineItem2Cost take no arguments ()
-- and returns a double (D)
CREATE FUNCTION getLineItem2Cost()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.getLineItem2Cost()D'
LANGUAGE JAVA;
```

The descriptors for arguments to and return values from Java methods have the following meanings:

Field type	Java data type
B	byte
C	char
D	double
F	float

Field type	Java data type
I	int
J	long
L *class-name*;	An instance of the class *class-name*. The class name must be fully qualified, and any dot in the name must be replaced by a /. For example, **java/lang/String**.
S	short
V	void
Z	Boolean
[	Use one for each dimension of an array.

For more information about the syntax of these statements, see "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*] and "CREATE FUNCTION statement" [*SQL Anywhere Server - SQL Reference*].

2. Call the stored procedure that is acting as a wrapper to call the Java method:

```
CALL init('Shirt',10.00,'Jacket',25.00);
SELECT getLineItem1Description() as Item1,
       getLineItem1Cost() as Item1Cost,
       getLineItem2Description() as Item2,
       getLineItem2Cost() as Item2Cost,
       rateOfTaxation() as TaxRate,
       totalSum() as Cost;
```

The query returns six columns with values as follows:

Item1	Item1Cost	Item2	Item2Cost	TaxRate	Cost
Shirt	10	Jacket	25	0.15	40.25

Installing Java classes into a database

You can install Java classes into a database as:

- **A single class** You can install a single class into a database from a compiled class file. Class files typically have extension *.class*.
- **A JAR** You can install a set of classes all at once if they are in either a compressed or uncompressed JAR file. JAR files typically have the extension *.jar* or *.zip*. SQL Anywhere supports all compressed JAR files created with the Sun JAR utility, as well as some other JAR compression schemes.

Creating a class

Although the details of each step may differ depending on whether you are using a Java development tool, the steps involved in creating your own class generally include the following:

To create a class

1. Define your class.

   Write the Java code that defines your class. If you are using the Sun Java SDK then you can use a text editor. If you are using a development tool, the development tool provides instructions.

   Use only supported classes
   User classes must be 100% Java. Native methods are not allowed.

2. Name and save your class.

   Save your class declaration (Java code) in a file with the extension *.java*. Make certain the name of the file is the same as the name of the class and that the case of both names is identical.

   For example, a class called Utility should be saved in a file called *Utility.java*.

3. Compile your class.

   This step turns your class declaration containing Java code into a new, separate file containing byte code. The name of the new file is the same as the Java code file, but has an extension of *.class*. You can run a compiled Java class in a Java runtime environment, regardless of the platform you compiled it on or the operating system of the runtime environment.

   The Sun JDK contains a Java compiler, *javac*.

Installing a class

To make your Java class available within the database, you install the class into the database either from Sybase Central, or using the INSTALL JAVA statement from Interactive SQL or another application. You must know the path and file name of the class you want to install.

You require DBA authority to install a class.

To install a class (Sybase Central)

1. Connect to a database as a DBA user.
2. Open the **External Environments** folder.
3. Under this folder, open the **Java** folder.
4. Right-click the right pane and choose **New** » **Java Class** from the popup menu.
5. Follow the instructions in the wizard.

To install a class (SQL)

1. Connect to the database as a DBA user.
2. Execute the following statement:

```
INSTALL JAVA NEW
FROM FILE 'path\\ClassName.class';
```

   path is the directory where the class file is located, and *ClassName.class* is the name of the class file.

   The double backslash ensures that the backslash is not treated as an escape character.

   For example, to install a class in a file named *Utility.class*, held in the directory *c:\source*, you would execute the following statement:

```
INSTALL JAVA NEW
FROM FILE 'c:\\source\\Utility.class';
```

   If you use a relative path, it must be relative to the current working directory of the database server.

   For more information, see "INSTALL JAVA statement" [*SQL Anywhere Server - SQL Reference*].

Installing a JAR

It is useful and common practice to collect sets of related classes together in packages, and to store one or more packages in a **JAR file**.

You install a JAR file the same way as you install a class file. A JAR file can have the extension JAR or ZIP. Each JAR file must have a name in the database. Usually, you use the same name as the JAR file, without the extension. For example, if you install a JAR file named *myjar.zip*, you would generally give it a JAR name of *myjar*.

For more information, see "INSTALL JAVA statement" [*SQL Anywhere Server - SQL Reference*].

To install a JAR (Sybase Central)

1. Connect to the database as a DBA user.
2. Open the **External Environments** folder.
3. Under this folder, open the **Java** folder.
4. Right-click the right pane and choose **New** » **JAR File** from the popup menu.

5. Follow the instructions in the wizard.

To install a JAR (SQL)

1. Connect to a database as a DBA user.
2. Execute the following statement:

```
INSTALL JAVA NEW
JAR 'jarname'
FROM FILE 'path\\JarName.jar';
```

Updating classes and JAR files

You can update classes and JAR files using Sybase Central or by executing an INSTALL JAVA statement from Interactive SQL or some other client application.

To update a class or JAR, you must have DBA authority and a newer version of the compiled class file or JAR file available in a file on disk.

When updated classes take effect

Only new connections established after installing the class, or that use the class for the first time after installing the class, use the new definition. Once the Java VM loads a class definition, it stays in memory until the connection closes.

If you have been using a Java class or objects based on a class in the current connection, you need to disconnect and reconnect to use the new class definition.

To update a class or JAR (Sybase Central)

1. Connect to the database as a DBA user.
2. Open the **External Environments** folder.
3. Under this folder, open the **Java** folder.
4. Locate the subfolder containing the class or JAR file you want to update.
5. Select the class or JAR file and the choose **File** » **Update**.

   The **Update** window appears.
6. In the **Update** window, specify the name and location of the class or JAR file to be updated. You can click **Browse** to search for it.

Tips

You can also update a Java class or JAR file by right-clicking the class or JAR file name and choosing **Update**.

As well, you can update a Java class or JAR file by clicking **Update Now** on the **General** tab of its **Properties** window.

To update a class or JAR (SQL)

1. Connect to a database as a DBA user.
2. Execute the following statement:

```
INSTALL JAVA UPDATE
[ JAR 'jarname' ]
FROM FILE 'filename';
```

If you are updating a JAR, you must enter the name by which the JAR is known in the database. See "INSTALL JAVA statement" [*SQL Anywhere Server - SQL Reference*].

Special features of Java classes in the database

This section describes features of Java classes when used in the database.

Calling the main method

You typically start Java applications (outside the database) by running the Java VM on a class that has a main method.

For example, the Invoice class in the file *samples-dir\SQLAnywhere\JavaInvoice\Invoice.java* has a main method. When you execute the class from the command line using a command such as the following, it is the main method that executes:

```
java Invoice
```

To call the main method of a class from SQL

1. Declare the method with an array of strings as an argument:

   ```
   public static void main( java.lang.String args[] )
   {
   ...
   }
   ```

2. Create a stored procedure that wraps this method.

   ```
   CREATE PROCEDURE JavaMain( in arg char(50) )
   EXTERNAL NAME 'JavaClass.main([Ljava/lang/String;)V'
   LANGUAGE JAVA;
   ```

   For more information, see "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*].

3. Invoke the main method using the CALL statement.

   ```
   call JavaMain( 'Hello world' );
   ```

   Due to the limitations of the SQL language, only a single string can be passed.

Using threads in Java applications

With features of the java.lang.Thread package, you can use multiple threads in a Java application.

You can synchronize, suspend, resume, interrupt, or stop threads in Java applications.

No Such Method Exception

If you supply an incorrect number of arguments when calling a Java method, or if you use an incorrect data type, the Java VM responds with a `java.lang.NoSuchMethodException` error. You should check the number and type of arguments.

For more information, see "Accessing fields and methods of the Java object" on page 92.

Returning result sets from Java methods

This section describes how to make result sets available from Java methods. You must write a Java method that returns a result set to the calling environment, and wrap this method in a SQL stored procedure declared to be EXTERNAL NAME of LANGUAGE JAVA.

To return result sets from a Java method

1. Ensure that the Java method is declared as public and static in a public class.
2. For each result set you expect the method to return, ensure that the method has a parameter of type java.sql.ResultSet[]. These result set parameters must all occur at the end of the parameter list.
3. In the method, first create an instance of java.sql.ResultSet and then assign it to one of the ResultSet[] parameters.
4. Create a SQL stored procedure of type EXTERNAL NAME LANGUAGE JAVA. This type of procedure is a wrapper around a Java method. You can use a cursor on the SQL procedure result set in the same way as any other procedure that returns result sets.

   For more information about the syntax for stored procedures that are wrappers for Java methods, see "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*].

Example

The following simple class has a single method that executes a query and passes the result set back to the calling environment.

```
import java.sql.*;

public class MyResultSet
{
  public static void return_rset( ResultSet[] rset1 )
      throws SQLException
  {
    Connection conn = DriverManager.getConnection(
                      "jdbc:default:connection" );
    Statement stmt = conn.createStatement();
    ResultSet rset =
      stmt.executeQuery (
                      "SELECT Surname " +
                      "FROM Customers" );
    rset1[0] = rset;
  }
}
```

You can expose the result set using a CREATE PROCEDURE statement that indicates the number of result sets returned from the procedure and the signature of the Java method.

A CREATE PROCEDURE statement indicating a result set could be defined as follows:

```
CREATE PROCEDURE result_set()
  DYNAMIC RESULT SETS 1
  EXTERNAL NAME
```



```
   'MyResultSet.return_rset([Ljava/sql/ResultSet;)V'
LANGUAGE JAVA
```

You can open a cursor on this procedure, just as you can with any SQL Anywhere procedure returning result sets.

The string `([Ljava/sql/ResultSet;)V` is a Java method signature that is a compact character representation of the number and type of the parameters and return value.

For more information about Java method signatures, see "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*].

For more information about returning result sets, see "Returning result sets" on page 499.

Returning values from Java via stored procedures

You can use stored procedures created using the EXTERNAL NAME LANGUAGE JAVA as wrappers around Java methods. This section describes how to write your Java method to exploit OUT or INOUT parameters in the stored procedure.

Java does not have explicit support for INOUT or OUT parameters. Instead, you can use an array of the parameter. For example, to use an integer OUT parameter, create an array of exactly one integer:

```
public class Invoice
{
  public static boolean testOut( int[] param )
  {
    param[0] = 123;
    return true;
  }
}
```

The following procedure uses the testOut method:

```
CREATE PROCEDURE testOut( OUT p INTEGER )
EXTERNAL NAME 'Invoice.testOut([I)Z'
LANGUAGE JAVA;
```

The string `([I)Z` is a Java method signature, indicating that the method has a single parameter, which is an array of integers, and returns a Boolean value. You must define the method so that the method parameter you want to use as an OUT or INOUT parameter is an array of a Java data type that corresponds to the SQL data type of the OUT or INOUT parameter.

To test this, call the stored procedure with an uninitialized variable.

```
CREATE VARIABLE zap INTEGER;
CALL testOut( zap );
SELECT zap;
```

The result set is `123`.

For more information about the syntax, including the method signature, see "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*].

Security management for Java

Java provides security managers than you can use to control user access to security-sensitive features of your applications, such as file access and network access. You should take advantage of the security management features supported by your Java VM.

Starting and stopping the Java VM

The Java VM loads automatically whenever the first Java operation is carried out. If you want to load it explicitly in readiness for carrying out Java operations, you can do so by executing the following statement:

```
START JAVA;
```

You can unload the Java VM when Java is not in use using the STOP JAVA statement. Only a user with DBA authority can execute this statement. The syntax is:

```
STOP JAVA;
```

Unsupported Java classes

You cannot use all classes from the JDK. The runtime Java classes available for use in the database server belong to a subset of the Java API. Classes in the following packages are not supported in SQL Anywhere:

- java.applet
- java.awt
- java.awt.datatransfer
- java.awt.event
- java.awt.image
- All packages prefixed by `sun` (for example, `sun.audio`)

Part III. SQL Anywhere Data Access APIs

This part describes the programming interfaces for SQL Anywhere.

CHAPTER 7

SQL Anywhere .NET Data Provider

Contents

SQL Anywhere .NET Data Provider features

SQL Anywhere supports Microsoft's .NET Framework version 2.0 or later through three distinct namespaces.

- **iAnywhere.Data.SQLAnywhere** The ADO.NET object model is an all-purpose data access model. ADO.NET components were designed to factor data access from data manipulation. There are two central components of ADO.NET that accomplish this: the DataSet, and the .NET Framework data provider, which is a set of components including the Connection, Command, DataReader, and DataAdapter objects. SQL Anywhere includes a .NET Framework Data Provider that communicates directly with a SQL Anywhere database server without adding the overhead of OLE DB or ODBC. The SQL Anywhere .NET Data Provider is represented in the .NET namespace as iAnywhere.Data.SQLAnywhere.

  The Microsoft .NET Compact Framework is the smart device development framework for Microsoft .NET. The SQL Anywhere .NET Compact Framework Data Provider supports devices running Windows Mobile.

  The SQL Anywhere .NET Data Provider namespace is described in this document.
- **System.Data.Oledb** This namespace supports OLE DB data sources. This namespace is an intrinsic part of the Microsoft .NET Framework. You can use System.Data.Oledb together with the SQL Anywhere OLE DB provider, SAOLEDB, to access SQL Anywhere databases.
- **System.Data.Odbc** This namespace supports ODBC data sources. This namespace is an intrinsic part of the Microsoft .NET Framework. You can use System.Data.Odbc together with the SQL Anywhere ODBC driver to access SQL Anywhere databases.

On Windows Mobile, only the SQL Anywhere .NET Data Provider is supported.

There are some key benefits to using the SQL Anywhere .NET Data Provider:

- In the .NET environment, the SQL Anywhere .NET Data Provider provides native access to a SQL Anywhere database. Unlike the other supported providers, it communicates directly with a SQL Anywhere server and does not require bridge technology.
- As a result, the SQL Anywhere .NET Data Provider is faster than the OLE DB and ODBC Data Providers. It is the recommended Data Provider for accessing SQL Anywhere databases.

Running the sample projects

There are four sample projects included with the SQL Anywhere .NET Data Provider:

- **SimpleCE** A .NET Compact Framework sample project for Windows Mobile that demonstrates a simple listbox that is filled with the names from the Employees table when you click the Connect button.
- **SimpleWin32** A .NET Framework sample project for Windows that demonstrates a simple listbox that is filled with the names from the Employees table when you click the Connect button.
- **SimpleXML** A .NET Framework sample project for Windows that demonstrates how to obtain XML data from SQL Anywhere via ADO.NET.
- **TableViewer** A .NET Framework sample project for Windows that allows you to enter and execute SQL statements.

For tutorials explaining the sample projects, see "Tutorial: Using the SQL Anywhere .NET Data Provider" on page 143.

Note

If you installed SQL Anywhere in a location other than the default installation directory (*C:\Program Files\SQL Anywhere 11*), you may receive an error referencing the Data Provider DLL when you load the sample projects. If this happens, add a new reference to *iAnywhere.Data.SQLAnywhere.dll*. There is one version of the Data Provider that supports the .NET Framework 2.0 and later versions. The Data Provider for Windows is located in *install-dir\Assembly\v2\iAnywhere.Data.SQLAnywhere.dll*. The Data Provider for Windows Mobile is located in *install-dir\ce\Assembly*.

For instructions on adding a reference to the DLL, see "Adding a reference to the Data Provider DLL in your project" on page 110.

Using the .NET Data Provider in a Visual Studio project

The SQL Anywhere .NET Data Provider can be used to develop applications with Visual Studio 2005 or later versions. To use the SQL Anywhere .NET Data Provider, you must include two items in your Visual Studio project:

- a reference to the SQL Anywhere .NET Data Provider DLL
- a line in your source code referencing the SQL Anywhere .NET Data Provider classes

These steps are explained below.

For information about installing and registering the SQL Anywhere .NET Data Provider, see "Deploying the SQL Anywhere .NET Data Provider" on page 137.

Adding a reference to the Data Provider DLL in your project

Adding a reference tells Visual Studio which DLL to include to find the code for the SQL Anywhere .NET Data Provider.

To add a reference to the SQL Anywhere .NET Data Provider in a Visual Studio project

1. Start Visual Studio and open your project.
2. In the **Solution Explorer** window, right-click **References** and choose **Add Reference** from the popup menu.

   The **Add Reference** window appears.
3. On the **.NET** tab, click **Browse** to locate *iAnywhere.Data.SQLAnywhere.dll*. Note that there are separate versions of the DLL for each of Windows and Windows Mobile platforms.
   - For the Windows SQL Anywhere .NET Data Provider, the default location is *install-dir\Assembly\v2*.
   - For the Windows Mobile SQL Anywhere .NET Data Provider, the default location is *install-dir\ce\Assembly\v2*.
4. Select the DLL and then click **Open**.

   For a complete list of installed DLLs, see "SQL Anywhere .NET Data Provider required files" on page 137.
5. You can verify that the DLL is added to your project. Open the **Add Reference** window and then click the **.NET** tab. *iAnywhere.Data.SQLAnywhere.dll* appears in the **Selected Components** list. Click **OK** to close the window.

   The DLL is added to the **References** folder in the **Solution Explorer** window of your project.

Using the Data Provider classes in your source code

To facilitate the use of the SQL Anywhere .NET Data Provider namespace and the types defined in this namespace, you should add a directive to your source code.

To facilitate the use of Data Provider namespace in your code

1. Start Visual Studio and open your project.
2. Add the following line to your project:
   - If you are using C#, add the following line to the list of `using` directives at the beginning of your project:

     ```
     using iAnywhere.Data.SQLAnywhere;
     ```
   - If you are using Visual Basic, add the following line at the beginning of your project before the line `Public Class Form1`:

     ```
     Imports iAnywhere.Data.SQLAnywhere
     ```

This directive is not required, however, it allows you to use short forms for the SQL Anywhere .NET classes. For example:

```
SAConnection conn = new SAConnection()
```

Without this directive, you can still use the following:

```
iAnywhere.Data.SQLAnywhere.SAConnection
 conn = new iAnywhere.Data.SQLAnywhere.SAConnection()
```

Connecting to a database

Before you can perform any operations on the data, your application must connect to the database. This section describes how to write code to connect to a SQL Anywhere database.

For more information, see "SAConnectionStringBuilder class" on page 249 and "ConnectionName property" on page 258.

To connect to a SQL Anywhere database

1. Allocate an SAConnection object.

   The following code creates an SAConnection object named conn:

   ```
   SAConnection conn = new SAConnection(connection-string)
   ```

   You can have more than one connection to a database from your application. Some applications use a single connection to a SQL Anywhere database, and keep the connection open all the time. To do this, you can declare a global variable for the connection:

   ```
   private SAConnection  _conn;
   ```

   For more information, see the sample code in *samples-dir\SQLAnywhere\ADO.NET\TableViewer* and "Understanding the Table Viewer sample project" on page 149.

2. Specify the connection string used to connect to the database.

   For example:

   ```
   "Data Source=SQL Anywhere 11 Demo"
   ```

   For a complete list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

   Instead of supplying a connection string, you can prompt users for their user ID and password.

3. Open a connection to the database.

   The following code attempts to connect to a database. It autostarts the database server if necessary.

   ```
   conn.Open();
   ```

4. Catch connection errors.

   Your application should be designed to catch any errors that occur when attempting to connect to the database. The following code demonstrates how to catch an error and display its message:

   ```
   try {
       _conn = new SAConnection( txtConnectString.Text );
       _conn.Open();
     } catch( SAException ex ) {
       MessageBox.Show( ex.Errors[0].Source + " : "
        + ex.Errors[0].Message + " (" +
        ex.Errors[0].NativeError.ToString() + ")",
            "Failed to connect" );
   ```

   Alternately, you can use the ConnectionString property to set the connection string, rather than passing the connection string when the SAConnection object is created:


```
SAConnection _conn;
    _conn = new SAConnection();
    _conn.ConnectionString =
     "Data Source=SQL Anywhere 11 Demo";
    _conn.Open();
```

5. Close the connection to the database. Connections to the database stay open until they are explicitly closed using the `conn.Close()` method.

Visual Basic connection example

The following Visual Basic code opens a connection to the SQL Anywhere sample database:

```
Private Sub Button1_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles Button1.Click
    ' Declare the connection object
    Dim myConn As New _
      iAnywhere.Data.SQLAnywhere.SAConnection()
    myConn.ConnectionString = _
      "Data Source=SQL Anywhere 11 Demo"
    myConn.Open()
    myConn.Close()
End Sub
```

Connection pooling

The SQL Anywhere .NET Data Provider supports connection pooling. Connection pooling allows your application to reuse existing connections by saving the connection handle to a pool so it can be reused, rather than repeatedly creating a new connection to the database. Connection pooling is turned on by default.

The pool size is set in your connection string using the POOLING option. The default maximum pool size is 100, while the default minimum pool size is 0. You can specify the minimum and maximum pool sizes. For example:

```
"Data Source=SQL Anywhere 11 Demo;POOLING=TRUE;Max Pool Size=50;Min Pool
Size=5"
```

When your application first attempts to connect to the database, it checks the pool for an existing connection that uses the same connection parameters you have specified. If a matching connection is found, that connection is used. Otherwise, a new connection is used. When you disconnect, the connection is returned to the pool so that it can be reused.

See also

- “ConnectionName property” on page 258
- “AutoStop connection parameter [ASTOP]” [*SQL Anywhere Server - Database Administration*]

Checking the connection state

Once your application has established a connection to the database, you can check the connection state to ensure that the connection is open before you fetch data from the database to update it. If a connection is lost or is busy, or if another statement is being processed, you can return an appropriate message to the user.

The SAConnection class has a state property that checks the state of the connection. Possible state values are Open and Closed.

The following code checks whether the Connection object has been initialized, and if it has, it ensures that the connection is open. A message is returned to the user if the connection is not open.

```
if( _conn == null || _conn.State !=
  ConnectionState.Open ) {
  MessageBox.Show( "Connect to a database first",
   "Not connected" );
  return;
```

For more information, see "State property" on page 238.

Accessing and manipulating data

With the SQL Anywhere .NET Data Provider, there are two ways you can access data:

- **SACommand object** The SACommand object is the recommended way of accessing and manipulating data in .NET.

  The SACommand object allows you to execute SQL statements that retrieve or modify data directly from the database. Using the SACommand object, you can issue SQL statements and call stored procedures directly against the database.

  Within an SACommand object, an SADataReader is used to return read-only result sets from a query or stored procedure. The SADataReader returns only one row at a time, but this does not degrade performance because the SQL Anywhere client-side libraries use prefetch buffering to prefetch several rows at a time.

  Using the SACommand object allows you to group your changes into transactions rather than operating in autocommit mode. When you use the SATransaction object, locks are placed on the rows so that other users cannot modify them.

  For more information, see "SACommand class" on page 192 and "SADataReader class" on page 289.

- **SADataAdapter object** The SADataAdapter object retrieves the entire result set into a DataSet. A DataSet is a disconnected store for data that is retrieved from a database. You can then edit the data in the DataSet and when you are finished, the SADataAdapter object updates the database with the changes made to the DataSet. When you use the SADataAdapter, there is no way to prevent other users from modifying the rows in your DataSet. You need to include logic within your application to resolve any conflicts that may occur.

  For more information about conflicts, see "Resolving conflicts when using the SADataAdapter" on page 123.

  For more information about the SADataAdapter object, see "SADataAdapter class" on page 278.

There is no performance impact from using the SADataReader within an SACommand object to fetch rows from the database rather than the SADataAdapter object.

Using the SACommand object to retrieve and manipulate data

The following sections describe how to retrieve data and how to insert, update, or delete rows using the SADataReader.

Getting data using the SACommand object

The SACommand object allows you to execute a SQL statement or call a stored procedure against a SQL Anywhere database. You can use any of the following methods to retrieve data from the database:

- **ExecuteReader** Issues a SQL query that returns a result set. This method uses a forward-only, read-only cursor. You can loop quickly through the rows of the result set in one direction.

  For more information, see "ExecuteReader methods" on page 211.

- **ExecuteScalar** Issues a SQL query that returns a single value. This can be the first column in the first row of the result set, or a SQL statement that returns an aggregate value such as COUNT or AVG. This method uses a forward-only, read-only cursor.

  For more information, see "ExecuteScalar method" on page 212.

When using the SACommand object, you can use the SADataReader to retrieve a result set that is based on a join. However, you can only make changes (inserts, updates, or deletes) to data that is from a single table. You cannot update result sets that are based on joins.

The following instructions use the Simple code sample included with the .NET Data Provider.

For more information about the Simple code sample, see "Understanding the Simple sample project" on page 146.

To issue a SQL query that returns a complete result set

1. Declare and initialize a Connection object.

   ```
   SAConnection conn = new SAConnection(
       "Data Source=SQL Anywhere 11 Demo" );
   ```

2. Open the connection.

   ```
   try {
       conn.Open();
   ```

3. Add a Command object to define and execute a SQL statement.

   ```
   SACommand cmd = new SACommand(
        "SELECT Surname FROM Employees", conn );
   ```

   If you are calling a stored procedure, you must specify the parameters for the stored procedure.

   For more information, see "Using stored procedures" on page 132 and "SAParameter class" on page 359.

4. Call the ExecuteReader method to return the DataReader object.

   ```
   SADataReader reader = cmd.ExecuteReader();
   ```

5. Display the results.

   ```
   listEmployees.BeginUpdate();
   while( reader.Read() ) {
        listEmployees.Items.Add( reader.GetString( 0 ) );
   }
   listEmployees.EndUpdate();
   ```


6. Close the DataReader and Connection objects.

```
reader.Close();
conn.Close();
```

To issue a SQL query that returns only one value

1. Declare and initialize an SAConnection object.

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 11 Demo" );
```

2. Open the connection.

```
conn.Open();
```

3. Add an SACommand object to define and execute a SQL statement.

```
SACommand cmd = new SACommand(
    "SELECT COUNT(*) FROM Employees WHERE Sex = 'M'",
    conn );
```

   If you are calling a stored procedure, you must specify the parameters for the stored procedure.

   For more information, see "Using stored procedures" on page 132.

4. Call the ExecuteScalar method to return the object containing the value.

```
int count = (int) cmd.ExecuteScalar();
```

5. Close the SAConnection object.

```
conn.Close();
```

When using the SADataReader, there are several Get methods available that you can use to return the results in the specified data type.

For more information, see "SADataReader class" on page 289.

Visual Basic DataReader example

The following Visual Basic code opens a connection to the SQL Anywhere sample database and uses the DataReader to return the last name of the first five employees in the result set:

```
Dim myConn As New .SAConnection()
Dim myCmd As _
  New .SACommand _
  ("SELECT Surname FROM Employees", myConn)
Dim myReader As SADataReader
Dim counter As Integer
myConn.ConnectionString = _
  "Data Source=SQL Anywhere 11 Demo"
myConn.Open()
myReader = myCmd.ExecuteReader()
counter = 0
Do While (myReader.Read())
  MsgBox(myReader.GetString(0))
  counter = counter + 1
  If counter >= 5 Then Exit Do
Loop
myConn.Close()
```

Inserting, updating, and deleting rows using the SACommand object

To perform an insert, update, or delete with the SACommand object, use the ExecuteNonQuery function. The ExecuteNonQuery function issues a query (SQL statement or stored procedure) that does not return a result set. See "ExecuteNonQuery method" on page 210.

You can only make changes (inserts, updates, or deletes) to data that is from a single table. You cannot update result sets that are based on joins. You must be connected to a database to use the SACommand object.

For information about obtaining primary key values for autoincrement primary keys, see "Obtaining primary key values" on page 127.

If you want to set the isolation level for a SQL statement, you must use the SACommand object as part of an SATransaction object. When you modify data without an SATransaction object, the .NET Data Provider operates in autocommit mode and any changes that you make are applied immediately. See "Transaction processing" on page 134.

To issue a statement that inserts a row

1. Declare and initialize an SAConnection object.

   ```
   SAConnection conn = new SAConnection(
       c_connStr );
   conn.Open();
   ```

2. Open the connection.

   ```
   conn.Open();
   ```

3. Add an SACommand object to define and execute an INSERT statement.

   You can use an INSERT, UPDATE, or DELETE statement with the ExecuteNonQuery method.

   ```
   SACommand insertCmd = new SACommand(
       "INSERT INTO Departments( DepartmentID, DepartmentName )
       VALUES( ?, ? )", conn );
   ```

   If you are calling a stored procedure, you must specify the parameters for the stored procedure.

   For more information, see "Using stored procedures" on page 132 and "SAParameter class" on page 359.

4. Set the parameters for the SACommand object.

   The following code defines parameters for the DepartmentID and DepartmentName columns respectively.

   ```
   SAParameter parm = new SAParameter();
   parm.SADbType = SADbType.Integer;
   insertCmd.Parameters.Add( parm );
   parm = new SAParameter();
   parm.SADbType = SADbType.Char;
   insertCmd.Parameters.Add( parm );
   ```

5. Insert the new values and call the ExecuteNonQuery method to apply the changes to the database.

   ```
   insertCmd.Parameters[0].Value = 600;
   insertCmd.Parameters[1].Value = "Eastern Sales";
   ```



```
int recordsAffected = insertCmd.ExecuteNonQuery();
insertCmd.Parameters[0].Value = 700;
insertCmd.Parameters[1].Value = "Western Sales";
recordsAffected = insertCmd.ExecuteNonQuery();
```

6. Display the results and bind them to the grid on the screen.

```
SACommand selectCmd = new SACommand(
    "SELECT * FROM Departments", conn );
SADataReader dr = selectCmd.ExecuteReader();

System.Windows.Forms.DataGrid dataGrid;
dataGrid = new System.Windows.Forms.DataGrid();
dataGrid.Location = new Point(10, 10);
dataGrid.Size = new Size(275, 200);
dataGrid.CaptionText = "iAnywhere SACommand Example";
this.Controls.Add(dataGrid);

dataGrid.DataSource = dr;
dataGrid.Show();
```

7. Close the SADataReader and SAConnection objects.

```
dr.Close();
conn.Close();
```

To issue a statement that updates a row

1. Declare and initialize an SAConnection object.

```
SAConnection conn = new SAConnection(
    c_connStr );
```

2. Open the connection.

```
conn.Open();
```

3. Add an SACommand object to define and execute an UPDATE statement.

   You can use an INSERT, UPDATE, or DELETE statement with the ExecuteNonQuery method.

```
SACommand updateCmd = new SACommand(
    "UPDATE Departments SET DepartmentName = 'Engineering'
    WHERE DepartmentID=100", conn );
```

   If you are calling a stored procedure, you must specify the parameters for the stored procedure.

   For more information, see "Using stored procedures" on page 132 and "SAParameter class" on page 359.

4. Call the ExecuteNonQuery method to apply the changes to the database.

```
int recordsAffected = updateCmd.ExecuteNonQuery();
```

5. Display the results and bind them to the grid on the screen.

```
SACommand selectCmd = new SACommand(
    "SELECT * FROM Departments", conn );
SADataReader   dr = selectCmd.ExecuteReader();
dataGrid.DataSource = dr;
```

6. Close the SADataReader and SAConnection objects.

```
dr.Close();
conn.Close();
```

To issue a statement that deletes a row

1. Declare and initialize an SAConnection object.

   ```
   SAConnection conn = new SAConnection(
       c_connStr );
   ```

2. Open the connection.

   ```
   conn.Open();
   ```

3. Create an SACommand object to define and execute a DELETE statement.

   You can use an INSERT, UPDATE, or DELETE statement with the ExecuteNonQuery method.

   ```
   SACommand deleteCmd = new SACommand(
       "DELETE FROM Departments WHERE ( DepartmentID > 500 )", conn );
   ```

   If you are calling a stored procedure, you must specify the parameters for the stored procedure.

   For more information, see "Using stored procedures" on page 132 and "SAParameter class" on page 359.

4. Call the ExecuteNonQuery method to apply the changes to the database.

   ```
   int recordsAffected = deleteCmd.ExecuteNonQuery();
   ```

5. Close the SAConnection object.

   ```
   conn.Close();
   ```

Obtaining DataReader schema information

You can obtain schema information about columns in the result set.

If you are using the SADataReader, you can use the GetSchemaTable method to obtain information about the result set. The GetSchemaTable method returns the standard .NET DataTable object, which provides information about all the columns in the result set, including column properties.

For more information about the GetSchemaTable method, see "GetSchemaTable method" on page 310.

To obtain information about a result set using the GetSchemaTable method

1. Declare and initialize a connection object.

   ```
   SAConnection conn = new SAConnection(
       c_connStr );
   ```

2. Open the connection.

   ```
   conn.Open();
   ```

3. Create an SACommand object with the SELECT statement you want to use. The schema is returned for the result set of this query.


```
SACommand cmd = new SACommand(
    "SELECT * FROM Employees", conn );
```

4. Create an SADataReader object and execute the Command object you created.

```
SADataReader   dr = cmd.ExecuteReader();
```

5. Fill the DataTable with the schema from the data source.

```
DataTable       schema = dr.GetSchemaTable();
```

6. Close the SADataReader and SAConnection objects.

```
dr.Close();
conn.Close();
```

7. Bind the DataTable to the grid on the screen.

```
dataGrid.DataSource = schema;
```

Using the SADataAdapter object to access and manipulate data

The following sections describe how to retrieve data and how to insert, update, or delete rows using the SADataAdapter.

Getting data using the SADataAdapter object

The SADataAdapter allows you to view the entire result set by using the Fill method to fill a DataSet with the results from a query by binding the DataSet to the display grid.

Using the SADataAdapter, you can pass any string (SQL statement or stored procedure) that returns a result set. When you use the SADataAdapter, all the rows are fetched in one operation using a forward-only, read-only cursor. Once all the rows in the result set have been read, the cursor is closed. The SADataAdapter allows you to make changes to the DataSet. Once your changes are complete, you must reconnect to the database to apply the changes.

You can use the SADataAdapter object to retrieve a result set that is based on a join. However, you can only make changes (inserts, updates, or deletes) to data that is from a single table. You cannot update result sets that are based on joins.

Caution
Any changes you make to the DataSet are made while you are disconnected from the database. This means that your application does not have locks on these rows in the database. Your application must be designed to resolve any conflicts that may occur when changes from the DataSet are applied to the database in the event that another user changes the data you are modifying before your changes are applied to the database.

For more information about the SADataAdapter, see “SADataAdapter class” on page 278.

SADataAdapter example

The following example shows how to fill a DataSet using the SADataAdapter.

To retrieve data using the SADataAdapter object

1. Connect to the database.
2. Create a new DataSet. In this case, the DataSet is called Results.

```
DataSet ds =new DataSet ();
```

3. Create a new SADataAdapter object to execute a SQL statement and fill the DataSet.

```
SADataAdapter da=new SADataAdapter(
     txtSQLStatement.Text, _conn);
da.Fill(ds, "Results")
```

4. Bind the DataSet to the grid on the screen.

```
dgResults.DataSource = ds.Tables["Results"]
```

Inserting, updating, and deleting rows using the SADataAdapter object

The SADataAdapter retrieves the result set into a DataSet. A DataSet is a collection of tables and the relationships and constraints between those tables. The DataSet is built into the .NET Framework, and is independent of the Data Provider used to connect to your database.

When you use the SADataAdapter, you must be connected to the database to fill the DataSet and to update the database with changes made to the DataSet. However, once the DataSet is filled, you can modify the DataSet while disconnected from the database.

If you do not want to apply your changes to the database right away, you can write the DataSet, including the data and/or the schema, to an XML file using the WriteXML method. Then, you can apply the changes at a later time by loading a DataSet with the ReadXML method.

For more information, see the .NET Framework documentation for WriteXML and ReadXML.

When you call the Update method to apply changes from the DataSet to the database, the SADataAdapter analyzes the changes that have been made and then invokes the appropriate statements, INSERT, UPDATE, or DELETE, as necessary. When you use the DataSet, you can only make changes (inserts, updates, or deletes) to data that is from a single table. You cannot update result sets that are based on joins. If another user has a lock on the row you are trying to update, an exception is thrown.

Caution

Any changes you make to the DataSet are made while you are disconnected. This means that your application does not have locks on these rows in the database. Your application must be designed to resolve any conflicts that may occur when changes from the DataSet are applied to the database in the event that another user changes the data you are modifying before your changes are applied to the database.

Resolving conflicts when using the SADataAdapter

When you use the SADataAdapter, no locks are placed on the rows in the database. This means there is the potential for conflicts to arise when you apply changes from the DataSet to the database. Your application should include logic to resolve or log conflicts that arise.

Some of the conflicts that your application logic should address include:

- **Unique primary keys** If two users insert new rows into a table, each row must have a unique primary key. For tables with autoincrement primary keys, the values in the DataSet may become out of sync with the values in the data source.

  For information about obtaining primary key values for autoincrement primary keys, see "Obtaining primary key values" on page 127.

- **Updates made to the same value** If two users modify the same value, your application should include logic to determine which value is correct.
- **Schema changes** If a user modifies the schema of a table you have updated in the DataSet, the update will fail when you apply the changes to the database.
- **Data concurrency** Concurrent applications should see a consistent set of data. The SADataAdapter does not place a lock on rows that it fetches, so another user can update a value in the database once you have retrieved the DataSet and are working offline.

Many of these potential problems can be avoided by using the SACommand, SADataReader, and SATransaction objects to apply changes to the database. The SATransaction object is recommended because it allows you to set the isolation level for the transaction and it places locks on the rows so that other users cannot modify them.

For more information about using transactions to apply your changes to the database, see "Inserting, updating, and deleting rows using the SACommand object" on page 118.

To simplify the process of conflict resolution, you can design your INSERT, UPDATE, or DELETE statement to be a stored procedure call. By including INSERT, UPDATE, and DELETE statements in stored procedures, you can catch the error if the operation fails. In addition to the statement, you can add error handling logic to the stored procedure so that if the operation fails the appropriate action is taken, such as recording the error to a log file, or trying the operation again.

To insert rows into a table using the SADataAdapter

1. Declare and initialize an SAConnection object.

   ```
   SAConnection   conn = new SAConnection(
       c_connStr );
   ```

2. Open the connection.

   ```
   conn.Open();
   ```

3. Create a new SADataAdapter object.

   ```
   SADataAdapter adapter = new SADataAdapter();
   adapter.MissingMappingAction =
       MissingMappingAction.Passthrough;
   adapter.MissingSchemaAction =
       MissingSchemaAction.Add;
   ```

4. Create the necessary SACommand objects and define any necessary parameters.

   The following code creates a SELECT and an INSERT statement and defines the parameters for the INSERT statement.

```
adapter.SelectCommand = new SACommand(
    "SELECT * FROM Departments", conn );
adapter.InsertCommand = new SACommand(
    "INSERT INTO Departments( DepartmentID, DepartmentName )
    VALUES( ?, ? )", conn );
adapter.InsertCommand.UpdatedRowSource =
    UpdateRowSource.None;
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add(
    parm );
parm = new SAParameter();
parm.SADbType = SADbType.Char;
parm.SourceColumn = "DepartmentName";
parm.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parm );
```

5. Fill the DataTable with the results of the SELECT statement.

```
DataTable dataTable = new DataTable( "Departments" );
int rowCount = adapter.Fill( dataTable );
```

6. Insert the new rows into the DataTable and apply the changes to the database.

```
DataRow row1 = dataTable.NewRow();
row1[0] = 600;
row1[1] = "Eastern Sales";
dataTable.Rows.Add( row1 );
DataRow row2 = dataTable.NewRow();
row2[0] = 700;
row2[1] = "Western Sales";
dataTable.Rows.Add( row2 );
recordsAffected = adapter.Update( dataTable );
```

7. Display the results of the updates.

```
dataTable.Clear();
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. Close the connection.

```
conn.Close();
```

To update rows using the SADataAdapter object

1. Declare and initialize an SAConnection object.

```
SAConnection conn = new SAConnection( c_connStr );
```

2. Open the connection.

```
conn.Open();
```

3. Create a new SADataAdapter object.

```
SADataAdapter adapter = new SADataAdapter();
adapter.MissingMappingAction =
```



```
    MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
    MissingSchemaAction.Add;
```

4. Create an SACommand object and define its parameters.

   The following code creates a SELECT and an UPDATE statement and defines the parameters for the UPDATE statement.

```
adapter.SelectCommand = new SACommand(
    "SELECT * FROM Departments WHERE DepartmentID > 500",
    conn );
adapter.UpdateCommand = new SACommand(
    "UPDATE Departments SET DepartmentName = ?
    WHERE DepartmentID = ?", conn );
adapter.UpdateCommand.UpdatedRowSource =
    UpdateRowSource.None;
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Char;
parm.SourceColumn = "DepartmentName";
parm.SourceVersion = DataRowVersion.Current;
adapter.UpdateCommand.Parameters.Add( parm );
parm = new SAParameter();
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Original;
adapter.UpdateCommand.Parameters.Add( parm );
```

5. Fill the DataTable with the results of the SELECT statement.

```
DataTable dataTable = new DataTable( "Departments" );
int rowCount = adapter.Fill( dataTable );
```

6. Update the DataTable with the updated values for the rows and apply the changes to the database.

```
foreach ( DataRow row in dataTable.Rows )
{
row[1] = ( string ) row[1] + "_Updated";
}
recordsAffected = adapter.Update( dataTable );
```

7. Bind the results to the grid on the screen.

```
dataTable.Clear();
adapter.SelectCommand.CommandText =
    "SELECT * FROM Departments";
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. Close the connection.

```
conn.Close();
```

To delete rows from a table using the SADataAdapter object

1. Declare and initialize an SAConnection object.

```
SAConnection conn = new SAConnection( c_connStr );
```

2. Open the connection.

```
conn.Open();
```

3. Create an SADataAdapter object.

```
SADataAdapter adapter = new SADataAdapter();
adapter.MissingMappingAction =
    MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
    MissingSchemaAction.AddWithKey;
```

4. Create the required SACommand objects and define any necessary parameters.

   The following code creates a SELECT and a DELETE statement and defines the parameters for the DELETE statement.

```
adapter.SelectCommand = new SACommand(
    "SELECT * FROM Departments WHERE DepartmentID > 500",
    conn );
adapter.DeleteCommand = new SACommand(
    "DELETE FROM Departments WHERE DepartmentID = ?",
    conn );
adapter.DeleteCommand.UpdatedRowSource =
    UpdateRowSource.None;
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Original;
adapter.DeleteCommand.Parameters.Add( parm );
```

5. Fill the DataTable with the results of the SELECT statement.

```
DataTable dataTable = new DataTable( "Departments" );
int rowCount = adapter.Fill( dataTable );
```

6. Modify the DataTable and apply the changes to the database.

```
for each ( DataRow in dataTable.Rows )
{
    row.Delete();
}
recordsAffected = adapter.Update( dataTable )
```

7. Bind the results to the grid on the screen.

```
dataTable.Clear();
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. Close the connection.

```
conn.Close();
```

Obtaining SADataAdapter schema information

When using the SADataAdapter, you can use the FillSchema method to obtain schema information about the result set in the DataSet. The FillSchema method returns the standard .NET DataTable object, which provides the names of all the columns in the result set.

To obtain DataSet schema information using the FillSchema method

1. Declare and initialize an SAConnection object.


```
SAConnection   conn = new SAConnection(
    c_connStr );
```

2. Open the connection.

```
conn.Open();
```

3. Create an SADataAdapter with the SELECT statement you want to use. The schema is returned for the result set of this query.

```
SADataAdapter  adapter = new SADataAdapter(
    "SELECT * FROM Employees", conn );
```

4. Create a new DataTable object, in this case called Table, to fill with the schema.

```
DataTable       dataTable = new DataTable(
    "Table" );
```

5. Fill the DataTable with the schema from the data source.

```
adapter.FillSchema( dataTable, SchemaType.Source );
```

6. Close the SAConnection object.

```
conn.Close();
```

7. Bind the DataSet to the grid on the screen.

```
dataGrid.DataSource = dataTable;
```

Obtaining primary key values

If the table you are updating has an autoincremented primary key, uses UUIDs, or if the primary key comes from a primary key pool, you can use a stored procedure to obtain values generated by the data source.

When using the SADataAdapter, this technique can be used to fill the columns in the DataSet with the primary key values generated by the data source. If you want to use this technique with the SACommand object, you can either get the key columns from the parameters or reopen the DataReader.

Examples

The following examples use a table called adodotnet_primarykey that contains two columns, ID and Name. The primary key for the table is ID. It is an INTEGER and contains an autoincremented value. The Name column is CHAR(40).

These examples call the following stored procedure to retrieve the autoincremented primary key value from the database.

```
CREATE PROCEDURE sp_adodotnet_primarykey( out p_id int, in p_name char(40) )
BEGIN
    INSERT INTO adodotnet_primarykey( name ) VALUES(
      p_name );
    SELECT @@IDENTITY INTO p_id;
END
```

To insert a new row with an autoincremented primary key using the SACommand object

1. Connect to the database.

```
SAConnection   conn = OpenConnection();
```

2. Create a new SACommand object to insert new rows into the DataTable. In the following code, the line `int id1 = ( int ) parmId.Value;` verifies the primary key value of the row.

```
SACommand      cmd = conn.CreateCommand();
cmd.CommandText = "sp_adodotnet_primarykey";
cmd.CommandType = CommandType.StoredProcedure;
SAParameter parmId = new SAParameter();
parmId.SADbType = SADbType.Integer;
parmId.Direction = ParameterDirection.Output;
cmd.Parameters.Add( parmId );
SAParameter parmName = new SAParameter();
parmName.SADbType = SADbType.Char;
parmName.Direction = ParameterDirection.Input;
cmd.Parameters.Add( parmName );
parmName.Value = "R & D --- Command";
cmd.ExecuteNonQuery();
int id1 = ( int ) parmId.Value;
parmName.Value = "Marketing --- Command";
cmd.ExecuteNonQuery();
int id2 = ( int ) parmId.Value;
parmName.Value = "Sales --- Command";
cmd.ExecuteNonQuery();
int id3 = ( int ) parmId.Value;
parmName.Value = "Shipping --- Command";
cmd.ExecuteNonQuery();
int id4 = ( int ) parmId.Value;
```

3. Bind the results to the grid on the screen and apply the changes to the database.

```
cmd.CommandText = "SELECT * FROM " +
    adodotnet_primarykey";
cmd.CommandType = CommandType.Text;
SADataReader dr = cmd.ExecuteReader();
dataGrid.DataSource = dr;
```

4. Close the connection.

```
conn.Close();
```

To insert a new row with an autoincremented primary key using the SADataAdapter object

1. Create a new SADataAdapter.

```
DataSet         dataSet = new DataSet();
SAConnection   conn = OpenConnection();
SADataAdapter  adapter = new SADataAdapter();
adapter.MissingMappingAction =
    MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
    MissingSchemaAction.AddWithKey;
```

2. Fill the data and schema of the DataSet. The SelectCommand is called by the SADataAdapter.Fill method to do this. You can also create the DataSet manually without using the Fill method and SelectCommand if you do not need the existing records.

```
adapter.SelectCommand = new SACommand( "select * from  +
adodotnet_primarykey", conn );
```

3. Create a new SACommand to obtain the primary key values from the database.

```
adapter.InsertCommand = new SACommand(
    "sp_adodotnet_primarykey", conn );
```



```
adapter.InsertCommand.CommandType =
    CommandType.StoredProcedure;
adapter.InsertCommand.UpdatedRowSource =
    UpdateRowSource.OutputParameters;
SAParameter parmId = new SAParameter();
parmId.SADbType = SADbType.Integer;
parmId.Direction = ParameterDirection.Output;
parmId.SourceColumn = "ID";
parmId.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parmId );
SAParameter parmName = new SAParameter();
parmName.SADbType = SADbType.Char;
parmName.Direction = ParameterDirection.Input;
parmName.SourceColumn = "name";
parmName.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parmName );
```

4. Fill the DataSet.

   ```
   adapter.Fill( dataSet );
   ```

5. Insert the new rows into the DataSet.

   ```
   DataRow row = dataSet.Tables[0].NewRow();
   row[0] = -1;
   row[1] = "R & D --- Adapter";
   dataSet.Tables[0].Rows.Add( row );
   row = dataSet.Tables[0].NewRow();
   row[0] = -2;
   row[1] = "Marketing --- Adapter";
   dataSet.Tables[0].Rows.Add( row );
   row = dataSet.Tables[0].NewRow();
   row[0] = -3;
   row[1] = "Sales --- Adapter";
   dataSet.Tables[0].Rows.Add( row );
   row = dataSet.Tables[0].NewRow();
   row[0] = -4;
   row[1] = "Shipping --- Adapter";
   dataSet.Tables[0].Rows.Add( row );
   ```

6. Apply the changes in the DataSet to the database. When the Update method is called, the primary key values are changed to the values obtained from the database.

   ```
   adapter.Update( dataSet );
   dataGrid.DataSource = dataSet.Tables[0];
   ```

   When you add new rows to the DataTable and call the Update method, the SADataAdapter calls the InsertCommand and maps the output parameters to the key columns for each new row. The Update method is called only once, but the InsertCommand is called by the Update method as many times as necessary for each new row being added.

7. Close the connection to the database.

   ```
   conn.Close();
   ```

Handling BLOBs

When fetching long string values or binary data, there are methods that you can use to fetch the data in pieces. For binary data, use the GetBytes method, and for string data, use the GetChars method. Otherwise, BLOB data is treated in the same manner as any other data you fetch from the database.

For more information, see "GetBytes method" on page 298 and "GetChars method" on page 300.

To issue a statement that returns a string using the GetChars method

1. Declare and initialize a Connection object.
2. Open the connection.
3. Add a Command object to define and execute a SQL statement.

   ```
   SACommand cmd = new SACommand(
       "SELECT int_col, blob_col FROM test", conn );
   ```

4. Call the ExecuteReader method to return the DataReader object.

   ```
   SADataReader reader = cmd.ExecuteReader();
   ```

   The following code reads the two columns from the result set. The first column is an integer (`GetInt32( 0 )`), while the second column is a LONG VARCHAR. GetChars is used to read 100 characters at a time from the LONG VARCHAR column.

   ```
   int length = 100;
   char[] buf = new char[ length ];
   int intValue;
   long dataIndex = 0;
   long charsRead = 0;
   long blobLength = 0;
   while( reader.Read() ) {
       intValue = reader.GetInt32( 0 );
       while ( ( charsRead = reader.GetChars(
           1, dataIndex, buf, 0, length ) ) == ( long )
               length ) {
       dataIndex += length;
   }
   blobLength = dataIndex + charsRead;
   }
   ```

5. Close the DataReader and Connection objects.

   ```
   reader.Close();
   conn.Close();
   ```

Obtaining time values

The .NET Framework does not have a Time structure. If you want to fetch time values from SQL Anywhere, you must use the GetTimeSpan method. Using this method returns the data as a .NET Framework TimeSpan object.

For more information about the GetTimeSpan method, see "GetTimeSpan method" on page 312.

To convert a time value using the GetTimeSpan method

1. Declare and initialize a connection object.

```
SAConnection conn = new SAConnection(
    "Data Source=dsn-time-test;UID=DBA;PWD=sql" );
```

2. Open the connection.

```
conn.Open();
```

3. Add a Command object to define and execute a SQL statement.

```
SACommand cmd = new SACommand(
    "SELECT ID, time_col FROM time_test", conn )
```

4. Call the ExecuteReader method to return the DataReader object.

```
SADataReader reader = cmd.ExecuteReader();
```

   The following code uses the GetTimeSpan method to return the time as TimeSpan.

```
while ( reader.Read() )
{
    int ID = reader.GetInt32();
    TimeSpan time = reader.GetTimeSpan();
}
```

5. Close the DataReader and Connection objects.

```
reader.Close();
conn.Close();
```

Using stored procedures

You can use stored procedures with the .NET Data Provider. The ExecuteReader method is used to call stored procedures that return a result set, while the ExecuteNonQuery method is used to call stored procedures that do not return a result set. The ExecuteScalar method is used to call stored procedures that return only a single value.

When you call a stored procedure, you must create an SAParameter object. Use a question mark as a placeholder for parameters, as follows:

```
sp_producttype( ?, ? )
```

For more information about the Parameter object, see "SAParameter class" on page 359.

To execute a stored procedure

1. Declare and initialize an SAConnection object.

   ```
   SAConnection conn = new SAConnection(
       "Data Source=SQL Anywhere 11 Demo" );
   ```

2. Open the connection.

   ```
   conn.Open();
   ```

3. Add an SACommand object to define and execute a SQL statement. The following code uses the CommandType property to identify the statement as a stored procedure.

   ```
   SACommand cmd = new SACommand( "ShowProductInfo",
       conn );
   cmd.CommandType = CommandType.StoredProcedure;
   ```

   If you do not specify the CommandType property, then you must use a question mark as a placeholder for parameters, as follows:

   ```
   SACommand cmd = new SACommand(
       "call ShowProductInfo(?)", conn );
   cmd.CommandType = CommandType.Text;
   ```

4. Add an SAParameter object to define the parameters for the stored procedure. You must create a new SAParameter object for each parameter the stored procedure requires.

   ```
   SAParameter param = cmd.CreateParameter();
   param.SADbType = SADbType.Int32;
   param.Direction = ParameterDirection.Input;
   param.Value = 301;
   cmd.Parameters.Add( param );
   ```

   For more information about the Parameter object, see "SAParameter class" on page 359.

5. Call the ExecuteReader method to return the DataReader object. The Get methods are used to return the results in the specified data type.

   ```
   SADataReader reader = cmd.ExecuteReader();
   reader.Read();
   int ID = reader.GetInt32(0);
   string name = reader.GetString(1);
   ```



```
string descrip = reader.GetString(2);
decimal price = reader.GetDecimal(6);
```

6. Close the SADataReader and SAConnection objects.

```
reader.Close();
conn.Close();
```

Alternative way to call a stored procedure

Step 3 in the above instructions presents two ways you can call a stored procedure. Another way you can call a stored procedure, without using a Parameter object, is to call the stored procedure from your source code, as follows:

```
SACommand cmd = new SACommand(
    "call ShowProductInfo( 301 )", conn );
```

For information about calling stored procedures that return a result set or a single value, see "Getting data using the SACommand object" on page 116.

For information about calling stored procedures that do not return a result set, see "Inserting, updating, and deleting rows using the SACommand object" on page 118.

Transaction processing

With the SQL Anywhere .NET Data Provider, you can use the SATransaction object to group statements together. Each transaction ends with a COMMIT or ROLLBACK, which either makes your changes to the database permanent or cancels all the operations in the transaction. Once the transaction is complete, you must create a new SATransaction object to make further changes. This behavior is different from ODBC and embedded SQL, where a transaction persists after you execute a COMMIT or ROLLBACK until the transaction is closed.

If you do not create a transaction, the SQL Anywhere .NET Data Provider operates in autocommit mode by default. There is an implicit COMMIT after each insert, update, or delete, and once an operation is completed, the change is made to the database. In this case, the changes cannot be rolled back.

For more information about the SATransaction object, see "SATransaction class" on page 416.

Setting the isolation level for transactions

The database isolation level is used by default for transactions. However, you can choose to specify the isolation level for a transaction using the IsolationLevel property when you begin the transaction. The isolation level applies to all statements executed within the transaction. The SQL Anywhere .NET Data Provider supports snapshot isolation.

For more information about isolation levels, see "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*].

The locks that SQL Anywhere uses when you enter a SELECT statement depend on the transaction's isolation level.

For more information about locking and isolation levels, see "Locking during queries" [*SQL Anywhere Server - SQL Usage*].

The following example uses an SATransaction object to issue and then roll back a SQL statement. The transaction uses isolation level 2 (RepeatableRead), which places a write lock on the row being modified so that no other database user can update the row.

To use an SATransaction object to issue a statement

1. Declare and initialize an SAConnection object.

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 11 Demo" );
```

2. Open the connection.

```
conn.Open();
```

3. Issue a SQL statement to change the price of Tee shirts.

```
string stmt = "UPDATE Products SET UnitPrice =
    2000.00 WHERE name = 'Tee shirt'";
```

4. Create an SATransaction object to issue the SQL statement using a Command object.

   Using a transaction allows you to specify the isolation level. Isolation level 2 (RepeatableRead) is used in this example so that another database user cannot update the row.


```
SATransaction trans = conn.BeginTransaction(
    IsolationLevel.RepeatableRead );
SACommand cmd = new SACommand( stmt, conn,
    trans );
int rows = cmd.ExecuteNonQuery();
```

5. Roll back the changes.

   ```
   trans.Rollback();
   ```

   The SATransaction object allows you to commit or roll back your changes to the database. If you do not use a transaction, the .NET Data Provider operates in autocommit mode and you cannot roll back any changes that you make to the database. If you want to make the changes permanent, you would use the following:

   ```
   trans.Commit();
   ```

6. Close the SAConnection object.

   ```
   conn.Close();
   ```

Distributed transaction processing

The .NET 2.0 framework introduced a new namespace System.Transactions, which contains classes for writing transactional applications. Client applications can create and participate in distributed transactions with one or multiple participants. Client applications can implicitly create transactions using the TransactionScope class. The connection object can detect the existence of an ambient transaction created by the TransactionScope and automatically enlist. The client applications can also create a CommittableTransaction and call the EnlistTransaction method to enlist. This feature is supported by the SQL Anywhere .NET 2.0 Data Provider. Distributed transaction has significant performance overhead. It is recommended that you use database transactions for non-distributed transactions.

Error handling and the SQL Anywhere .NET Data Provider

Your application must be designed to handle any errors that occur, including ADO.NET errors. ADO.NET errors are handled within your code in the same way that you handle other errors in your application.

The SQL Anywhere .NET Data Provider throws SAException objects whenever errors occur during execution. Each SAException object consists of a list of SAError objects, and these error objects include the error message and code.

Errors are different from conflicts. Conflicts arise when changes are applied to the database. Your application should include a process to compute correct values or to log conflicts when they arise.

For more information about handling conflicts, see "Resolving conflicts when using the SADataAdapter" on page 123.

.NET Data Provider error handling example

The following example is from the Simple sample project. Any errors that occur during execution and that originate with SQL Anywhere .NET Data Provider objects are handled by displaying them in a window. The following code catches the error and displays its message:

```
catch( SAException ex ) {
    MessageBox.Show( ex.Errors[0].Message );
}
```

Connection error handling example

The following example is from the Table Viewer sample project. If there is an error when the application attempts to connect to the database, the following code uses a try and catch block to catch the error and display its message:

```
try {
    _conn = new SAConnection( txtConnectString.Text );
    _conn.Open();
  } catch( SAException ex ) {
    MessageBox.Show( ex.Errors[0].Source + " : "
     + ex.Errors[0].Message + " (" +
     ex.Errors[0].NativeError.ToString() + ")",
         "Failed to connect" );
```

For more error handling examples, see "Understanding the Simple sample project" on page 146 and "Understanding the Table Viewer sample project" on page 149.

For more information about error handling, see "SAFactory class" on page 337 and "SAError class" on page 327.

Deploying the SQL Anywhere .NET Data Provider

The following sections describe how to deploy the SQL Anywhere .NET Data Provider.

SQL Anywhere .NET Data Provider system requirements

To use the SQL Anywhere .NET Data Provider, you must have the following installed on your computer or handheld device:

- The .NET Framework and/or .NET Compact Framework version 2.0 or later.
- Visual Studio 2005 or later, or a .NET language compiler, such as C# (required only for development).

SQL Anywhere .NET Data Provider required files

The SQL Anywhere .NET Data Provider consists of two DLLs for each platform.

Windows required file

For Windows (except Windows Mobile), the following DLL is required:

- *install-dir\Assembly\v2\iAnywhere.Data.SQLAnywhere.dll*

The file *iAnywhere.Data.SQLAnywhere.dll* is the DLL that is referenced by Visual Studio projects. The DLL is required for .NET Framework version 2.0 or later applications.

Windows Mobile required files

For Windows Mobile, the following DLL is required:

- *install-dir\ce\Assembly\v2\iAnywhere.Data.SQLAnywhere.dll*

The file *iAnywhere.Data.SQLAnywhere.dll* is the DLL that is referenced by Visual Studio projects. The DLL is required for .NET Compact Framework version 2.0 or later applications.

Visual Studio deploys the .NET Data Provider DLL (*iAnywhere.Data.SQLAnywhere.dll*) to your device along with your program. If you are not using Visual Studio, you need to copy the Data Provider DLL to the device along with your program. It can go in the same directory as your application, or in the Windows directory.

Registering the SQL Anywhere .NET Data Provider DLL

The SQL Anywhere .NET Data Provider DLL (*install-dir\Assembly\v2\iAnywhere.Data.SQLAnywhere.dll*) needs to be registered in the Global Assembly Cache on Windows (except Windows Mobile). The Global Assembly Cache lists all the registered programs on your computer. When you install the .NET Data Provider, the .NET Data Provider installation program registers it. On Windows Mobile, you do not need to register the DLL.

If you are deploying the .NET Data Provider, you must register the .NET Data Provider DLL (*install-dir \Assembly\v2\iAnywhere.Data.SQLAnywhere.dll*) using the **gacutil** utility that is included with the .NET Framework.

Tracing support

The SQL Anywhere .NET provider supports tracing using the .NET 2.0 or later tracing feature. Note that tracing is not supported on Windows Mobile.

By default, tracing is disabled. To enable tracing, specify the trace source in your application's configuration file. Here's an example of the configuration file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.diagnostics>
<sources>
 <source name="iAnywhere.Data.SQLAnywhere"
         switchName="SASourceSwitch"
         switchType="System.Diagnostics.SourceSwitch">
  <listeners>
   <add name="ConsoleListener"
        type="System.Diagnostics.ConsoleTraceListener"/>
   <add name="EventListener"
        type="System.Diagnostics.EventLogTraceListener"
        initializeData="MyEventLog"/>
   <add name="TraceLogListener"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="myTrace.log"
        traceOutputOptions="ProcessId, ThreadId, Timestamp"/>
   <remove name="Default"/>
  </listeners>
 </source>
</sources>
<switches>
 <add name="SASourceSwitch" value="All"/>
 <add name="SATraceAllSwitch" value="1" />
 <add name="SATraceExceptionSwitch" value="1" />
 <add name="SATraceFunctionSwitch" value="1" />
 <add name="SATracePoolingSwitch" value="1" />
 <add name="SATracePropertySwitch" value="1" />
</switches>
</system.diagnostics>
</configuration>
```

The trace configuration information is placed in the application's *bin\debug* folder under the name *app.exe.config*.

The `traceOutputOptions` that can be specified include the following:

- **Callstack** Write the call stack, which is represented by the return value of the Environment.StackTrace property.
- **DateTime** Write the date and time.
- **LogicalOperationStack** Write the logical operation stack, which is represented by the return value of the CorrelationManager.LogicalOperationStack property.
- **None** Do not write any elements.
- **ProcessId** Write the process identity, which is represented by the return value of the Process.Id property.

- **ThreadId** Write the thread identity, which is represented by the return value of the Thread.ManagedThreadId property for the current thread.
- **Timestamp** Write the timestamp, which is represented by the return value of the System.Diagnostics.Stopwatch.GetTimeStamp method.

You can limit what is traced by setting specific trace options. By default the trace option settings are all 0. The trace options that can be set include the following:

- **SATraceAllSwitch** The Trace All switch. When specified, all the trace options are enabled. You do not need to set any other options since they are all selected. You cannot disable individual options if you choose this option. For example, the following will not disable exception tracing.

  ```
  <add name="SATraceAllSwitch" value="1" />
  <add name="SATraceExceptionSwitch" value="0" />
  ```

- **SATraceExceptionSwitch** All exceptions are logged. Trace messages have the following form.

  ```
  <Type|ERR> message='message_text'[ nativeError=error_number]
  ```

  The nativeError=error_number text will only be displayed if there is an SAException object.

- **SATraceFunctionSwitch** All function scope entry/exits are logged. Trace messages have any of the following forms.

  ```
  enter_nnn <sa.class_name.method_name|API> [object_id#][parameter_names]
  leave_nnn
  ```

  The nnn is an integer representing the scope nesting level 1, 2, 3,... The optional parameter_names is a list of parameter names separated by spaces.

- **SATracePoolingSwitch** All connection pooling is logged. Trace messages have any of the following forms.

  ```
  <sa.ConnectionPool.AllocateConnection|CPOOL>
  connectionString='connection_text'
  <sa.ConnectionPool.RemoveConnection|CPOOL>
  connectionString='connection_text'
  <sa.ConnectionPool.ReturnConnection|CPOOL>
  connectionString='connection_text'
  <sa.ConnectionPool.ReuseConnection|CPOOL>
  connectionString='connection_text'
  ```

- **SATracePropertySwitch** All property setting and retrieval is logged. Trace messages have any of the following forms.

  ```
  <sa.class_name.get_property_name|API> object_id#
  <sa.class_name.set_property_name|API> object_id#
  ```

You can try application tracing using the TableViewer sample.

To configure an application for tracing

1. You must use .NET 2.0 or later.

   Start Visual Studio and open the TableViewer project file (*TableViewer.sln*) in *samples-dir\SQLAnywhere\ADO.NET\TableViewer*.

2. Place a copy of the configuration file shown above in the application's *bin\debug* folder under the name *TableViewer.exe.config*.
3. From the Debug menu, select Start Debugging.

When the application finishes execution, you will find a trace output file in *samples-dir\SQLAnywhere \ADO.NET\TableViewer\bin\Debug\myTrace.log*.

Tracing is not supported on Windows Mobile.

For more information, see "Tracing Data Access" at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnadonet/html/tracingdataaccess.asp.

CHAPTER 8

Tutorial: Using the SQL Anywhere .NET Data Provider

Contents

Introduction to the .NET Data Provider tutorial

This chapter explains how to use the Simple and Table Viewer sample projects that are included with the SQL Anywhere .NET Data Provider. The sample projects can be used with Visual Studio 2005 or later versions. The sample projects were developed with Visual Studio 2005. If you use a later version, you may have to run the Visual Studio **Upgrade Wizard**.

Using the Simple code sample

This tutorial is based on the Simple project that is included with SQL Anywhere.

The complete application can be found in your SQL Anywhere samples directory at *samples-dir\SQLAnywhere\ADO.NET\SimpleWin32*.

For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

The Simple project illustrates the following features:

- connecting to a database using the SAConnection object
- executing a query using the SACommand object
- obtaining the results using the SADataReader object
- basic error handling

For more information about how the sample works, see "Understanding the Simple sample project" on page 146.

To run the Simple code sample in Visual Studio

1. Start Visual Studio.
2. Choose **File** » **Open** » **Project**.
3. Browse to *samples-dir\SQLAnywhere\ADO.NET\SimpleWin32* and open the *Simple.sln* project.
4. When you use the SQL Anywhere .NET Data Provider in a project, you must add a reference to the Data Provider DLL. This has already been done in the Simple code sample. You can view the reference to the Data Provider DLL in the following location:
   - In the **Solution Explorer** window, open the **References** folder.
   - You should see iAnywhere.Data.SQLAnywhere in the list.

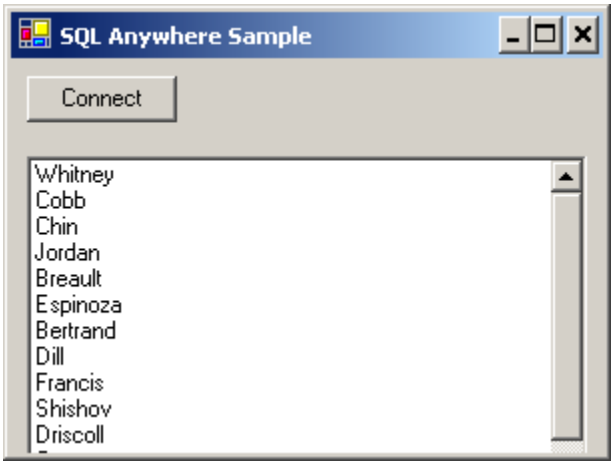
     For instructions about adding a reference to the Data Provider DLL, see "Adding a reference to the Data Provider DLL in your project" on page 110.
5. You must also add a `using` directive to your source code to reference the Data Provider classes. This has already been done in the Simple code sample. To view the `using` directive:
   - Open the source code for the project. In the **Solution Explorer** window, right-click *Form1.cs* and choose **View Code** from the popup menu.
   - In the `using` directives in the top section, you should see the following line:

     ```
     using iAnywhere.Data.SQLAnywhere;
     ```

     This line is required for C# projects. If you are using Visual Basic .NET, you need to add an `Imports` line to your source code.
6. Choose **Debug** » **Start Without Debugging** or press Ctrl+F5 to run the Simple sample.

   The **SQL Anywhere Sample** window appears.
   - In the **SQL Anywhere Sample** window, click **Connect**.

The application connects to the SQL Anywhere sample database and puts the last name of each employee in the window, as follows:



7. Click the **X** in the upper right corner of the screen to shut down the application and disconnect from the sample database. This also shuts down the database server.

You have now run the application. The next section describes the application code.

Understanding the Simple sample project

This section illustrates some key features of the SQL Anywhere .NET Data Provider by walking through some of the code from the Simple code sample. The Simple code sample uses the SQL Anywhere sample database, *demo.db*, which is held in your SQL Anywhere samples directory.

For information about the location of the SQL Anywhere samples directory, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

For information about the sample database, including the tables in the database and the relationships between them, see "SQL Anywhere sample database" [*SQL Anywhere 11 - Introduction*].

In this section, the code is described a few lines at a time. Not all code from the sample is included here. To see all the code, open the sample project in *samples-dir\SQLAnywhere\ADO.NET\SimpleWin32*.

Declaring controls The following code declares a button named btnConnect and a listbox named listEmployees.

```
private System.Windows.Forms.Button btnConnect;
private System.Windows.Forms.ListBox listEmployees;
```

Connecting to the database The btnConnect_Click method declares and initializes an SAConnection connection object.

```
private void btnConnect_Click(object sender,
   System.EventArgs e)
   SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 11 Demo;UID=DBA;PWD=sql" );
```


The SAConnection object uses the connection string to connect to the SQL Anywhere sample database when the Open method is called.

```
conn.Open();
```

For more information about the SAConnection object, see "SAConnection class" on page 231.

Defining a query A SQL statement is executed using an SACommand object. The following code declares and creates a command object using the SACommand constructor. This constructor accepts a string representing the query to be executed, along with the SAConnection object that represents the connection that the query is executed on.

```
SACommand cmd = new SACommand(
     "SELECT Surname FROM Employees", conn );
```

For more information about the SACommand object, see "SACommand class" on page 192.

Displaying the results The results of the query are obtained using an SADataReader object. The following code declares and creates an SADataReader object using the ExecuteReader constructor. This constructor is a member of the SACommand object, cmd, that was declared previously. ExecuteReader sends the command text to the connection for execution and builds an SADataReader.

```
SADataReader reader = cmd.ExecuteReader();
```

The following code loops through the rows held in the SADataReader object and adds them to the listbox control. Each time the Read method is called, the data reader gets another row back from the result set. A new item is added to the listbox for each row that is read. The data reader uses the GetString method with an argument of 0 to get the first column from the result set row.

```
listEmployees.BeginUpdate();
while( reader.Read() ) {
     listEmployees.Items.Add( reader.GetString( 0 ) );
}
listEmployees.EndUpdate();
```

For more information about the SADataReader object, see "SADataReader class" on page 289.

Finishing off The following code at the end of the method closes the data reader and connection objects.

```
reader.Close();
conn.Close();
```

Error handling Any errors that occur during execution and that originate with SQL Anywhere .NET Data Provider objects are handled by displaying them in a window. The following code catches the error and displays its message:

```
catch( SAException ex ) {
    MessageBox.Show( ex.Errors[0].Message );
}
```

For more information about the SAException object, see "SAException class" on page 333.

Using the Table Viewer code sample

This tutorial is based on the Table Viewer project that is included with the SQL Anywhere .NET Data Provider.

The complete application can be found in your SQL Anywhere samples directory in *samples-dir\SQLAnywhere\ADO.NET\TableViewer*.

For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

The Table Viewer project is more complex than the Simple project. It illustrates the following features:

- connecting to a database using the SAConnection object
- executing a query using the SACommand object
- obtaining the results using the SADataReader object
- using a grid to display the results using the DataGrid object
- more advanced error handling and result checking

For more information about how the sample works, see "Understanding the Table Viewer sample project" on page 149.

To run the Table Viewer code sample in Visual Studio

1. Start Visual Studio.
2. Choose **File** » **Open** » **Project**.
3. Browse to *samples-dir\SQLAnywhere\ADO.NET\TableViewer* and open the *TableViewer.sln* project.
4. If you want to use the SQL Anywhere .NET Data Provider in a project, you must add a reference to the Data Provider DLL. This has already been done in the Table Viewer code sample. You can view the reference to the Data Provider DLL in the following location:
   - In the **Solution Explorer** window, open the **References** folder.
   - You should see iAnywhere.Data.SQLAnywhere in the list.

     For instructions about adding a reference to the Data Provider DLL, see "Adding a reference to the Data Provider DLL in your project" on page 110.
5. You must also add a `using` directive to your source code to reference the Data Provider classes. This has already been done in the Table Viewer code sample. To view the `using` directive:
   - Open the source code for the project. In the **Solution Explorer** window, right-click *TableViewer.cs* and choose **View Code** from the popup menu.
   - In the `using` directives in the top section, you should see the following line:

     ```
     using iAnywhere.Data.SQLAnywhere;
     ```

     This line is required for C# projects. If you are using Visual Basic, you need to add an `Imports` line to your source code.
6. Choose **Debug** » **Start Without Debugging** or press Ctrl+F5 to run the Table Viewer sample.

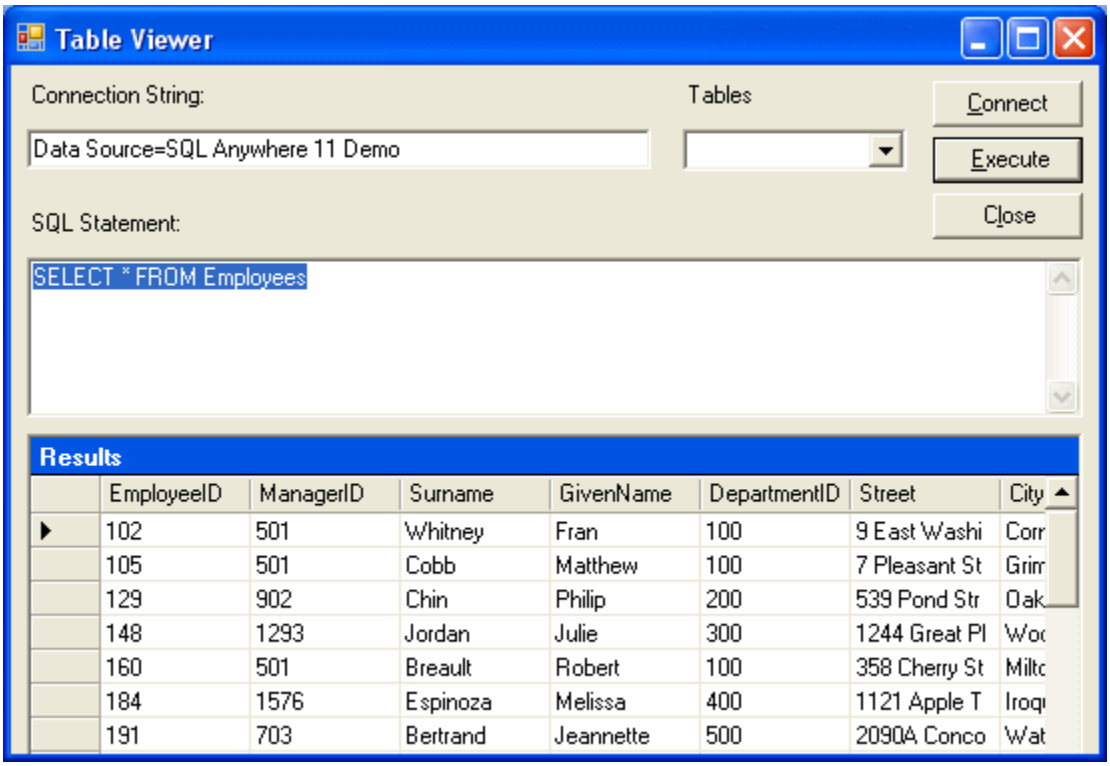
The **Table Viewer** window appears.

- In the **Table Viewer** window, click **Connect**.

  The application connects to the SQL Anywhere sample database.

- In the **Table Viewer** window, click **Execute**.

  The application retrieves the data from the Employees table in the sample database and puts the query results in the **Results** datagrid, as follows:



- You can also execute other SQL statements from this application: type a SQL statement in the **SQL Statement** pane, and then click **Execute**.

7. Click the **X** in the upper right corner of the screen to shut down the application and disconnect from the SQL Anywhere sample database. This also shuts down the database server.

You have now run the application. The next section describes the application code.

Understanding the Table Viewer sample project

This section illustrates some key features of the SQL Anywhere .NET Data Provider by walking through some of the code from the Table Viewer code sample. The Table Viewer project uses the SQL Anywhere sample database, *demo.db*, which is held in your SQL Anywhere samples directory.

For information about the location of the SQL Anywhere samples directory, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

For information about the sample database, including the tables in the database and the relationships between them, see "SQL Anywhere sample database" [*SQL Anywhere 11 - Introduction*].

In this section the code is described a few lines at a time. Not all code from the sample is included here. To see all the code, open the sample project in *samples-dir\SQLAnywhere\ADO.NET\TableViewer*.

Declaring controls The following code declares a couple of Labels named label1 and label2, a TextBox named txtConnectString, a button named btnConnect, a TextBox named txtSQLStatement, a button named btnExecute, and a DataGrid named dgResults.

```
private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox txtConnectString;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.Button btnConnect;
private System.Windows.Forms.TextBox txtSQLStatement;
private System.Windows.Forms.Button btnExecute;
private System.Windows.Forms.DataGrid dgResults;
```

Declaring a connection object The SAConnection type is used to declare an uninitialized SQL Anywhere connection object. The SAConnection object is used to represent a unique connection to a SQL Anywhere data source.

```
private SAConnection _conn;
```

For more information about the SAConnection class, see "SAConnection class" on page 231.

Connecting to the database The Text property of the txtConnectString object has a default value of "Data Source=SQL Anywhere 11 Demo". This value can be overridden by the application user by typing a new value into the txtConnectString text box. You can see how this default value is set by opening up the region or section in *TableViewer.cs* labeled Windows Form Designer Generated Code. In this section, you find the following line of code.

```
this.txtConnectString.Text = "Data Source=SQL Anywhere 11 Demo";
```

Later, the SAConnection object uses the connection string to connect to a database. The following code creates a new connection object with the connection string using the SAConnection constructor. It then establishes the connection by using the Open method.

```
_conn = new SAConnection( txtConnectString.Text );
_conn.Open();
```

For more information about the SAConnection constructor, see "SAConnection members" on page 232.

Defining a query The Text property of the txtSQLStatement object has a default value of "SELECT * FROM Employees". This value can be overridden by the application user by typing a new value into the txtSQLStatement text box.

The SQL statement is executed using an SACommand object. The following code declares and creates a command object using the SACommand constructor. This constructor accepts a string representing the query to be executed, along with the SAConnection object that represents the connection that the query is executed on.


```
SACommand cmd = new SACommand( txtSQLStatement.Text.Trim(),
                                          _conn );
```

For more information about the SACommand object, see "SACommand class" on page 192.

Displaying the results The results of the query are obtained using an SADataReader object. The following code declares and creates an SADataReader object using the ExecuteReader constructor. This constructor is a member of the SACommand object, cmd, that was declared previously. ExecuteReader sends the command text to the connection for execution and builds an SADataReader.

```
SADataReader dr = cmd.ExecuteReader();
```

The following code connects the SADataReader object to the DataGrid object, which causes the result columns to appear on the screen. The SADataReader object is then closed.

```
dgResults.DataSource = dr;
dr.Close();
```

For more information about the SADataReader object, see "SADataReader class" on page 289.

Error handling If there is an error when the application attempts to connect to the database or when it populates the Tables combo box, the following code catches the error and displays its message:

```
try {
  _conn = new SAConnection( txtConnectString.Text );
  _conn.Open();

  SACommand cmd = new SACommand(
    "SELECT table_name FROM SYS.SYSTAB where creator = 101", _conn );
  SADataReader   dr = cmd.ExecuteReader();

  comboBoxTables.Items.Clear();
  while ( dr.Read() ) {
    comboBoxTables.Items.Add( dr.GetString( 0 ) );
  }
  dr.Close();
} catch( SAException ex ) {
  MessageBox.Show( ex.Errors[0].Source + " : " +
       ex.Errors[0].Message + " (" +
       ex.Errors[0].NativeError.ToString() + ")",
       "Failed to connect" );
}
```

For more information about the SAException object, see "SAException class" on page 333.

CHAPTER 9

Tutorial: Developing a simple .NET database application with Visual Studio

Contents

Lesson 1. Create a table viewer

This tutorial is based on Visual Studio and the .NET Framework. The complete application can be found in the ADO.NET project *samples-dir\SQLAnywhere\ADO.NET\SimpleViewer\SimpleViewer.sln*.

In this tutorial, you use Microsoft Visual Studio, the Server Explorer, and the SQL Anywhere .NET Data Provider to create an application that accesses one of the tables in the SQL Anywhere sample database, allowing you to examine rows and perform updates.

To develop a database application with Visual Studio

1. Start Visual Studio.
2. From the Visual Studio **File** menu, choose **New** » **Project**.

   The **New Project** window appears.

   a. In the left pane of the **New Project** window, choose either **Visual Basic** or **Visual C#** for the programming language.

   b. From the **Windows** subcategory, choose **Windows Application** (VS 2005) or **Windows Forms Application** (VS 2008).

   c. In the project **Name** field, type **MySimpleViewer**.

   d. Click **OK** to create the new project.
3. In the Visual Studio **View** menu, choose **Server Explorer**.

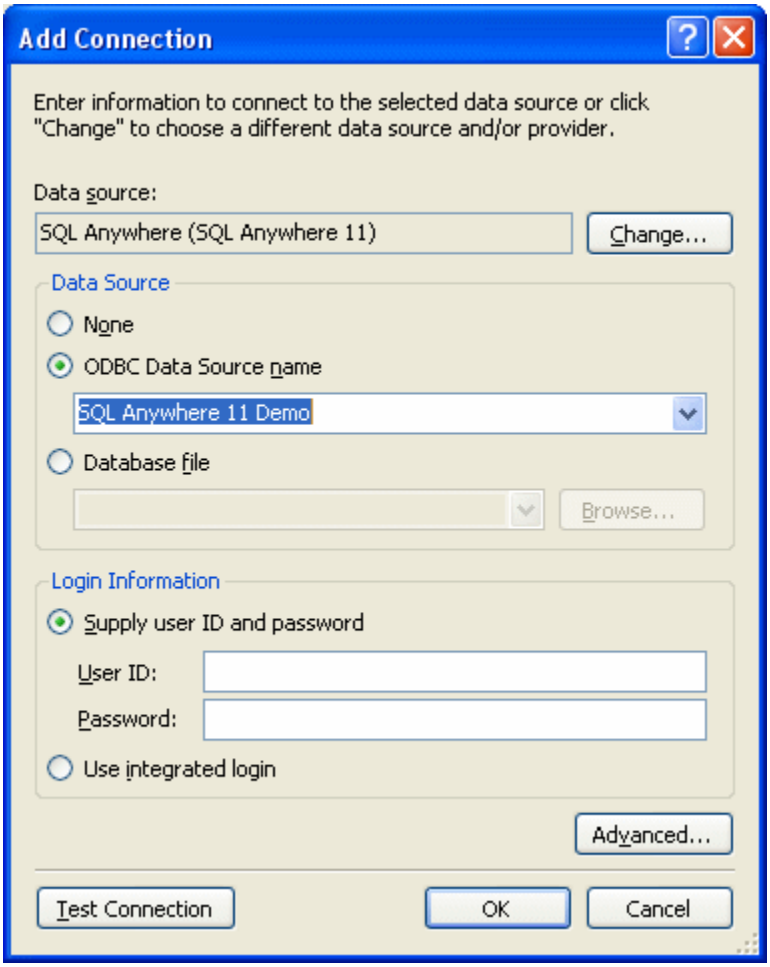
   The **Server Explorer** window appears.
4. In the **Server Explorer** window, right-click **Data Connections** and choose **Add Connection**.

   The **Add Connection** window appears.

   a. If you have never used **Add Connection** for other projects, then you will see a list of data sources. Select **SQL Anywhere** from the list of data sources presented.

      If you have used **Add Connection** before, then click **Change** to change the data source to **SQL Anywhere**.

   b. Under **Data Source**, select **ODBC Data Source Name** and type **SQL Anywhere 11 Demo**.



c. Click **Test Connection** to verify that you can connect to the sample database.

d. Click **OK**.

A new connection named **SQL Anywhere.demo11** appears in the **Server Explorer** window.

5. Expand the **SQL Anywhere.demo11** connection in the **Server Explorer** window until you see the table names.

   a. Right-click the Products table and choose **Show Table Data**.

      This shows the rows and columns of the Products table in a window.

   b. Close the table data window.

6. From the Visual Studio **Data** menu, choose **Add New Data Source**.

   The **Data Source Configuration Wizard** appears.

   a. On the **Data Source Type** page, select **Database** and click **Next**.

   b. On the **Data Connection** page, select **SQL Anywhere.demo11** and click **Next**.

c. On the **Save The Connection String** page, make sure that **Yes, Save The Connection As** is selected and click **Next**.

d. On the **Choose Your Database Objects** page, select **Tables** and click **Finish**.

7. From the Visual Studio **Data** menu, choose **Show Data Sources**.

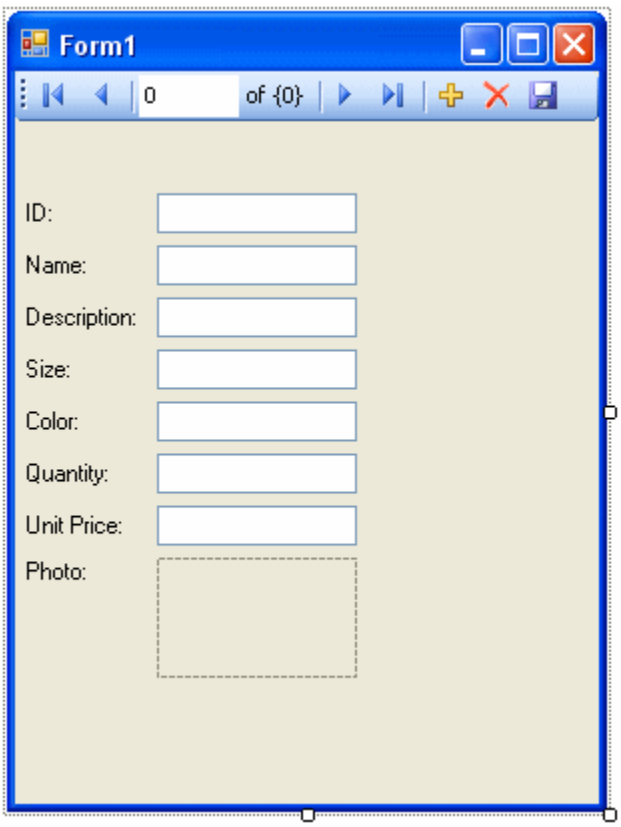
   The **Data Sources** window appears.

   Expand the Products table in the **Data Sources** window.

   a. Click Products and choose **Details** from the dropdown list.

   b. Click Photo and choose **Picture Box** from the dropdown list.

   c. Click Products and drag it to your form (Form1).



   A dataset control and several labeled text fields appear on the form.

8. On the form, select the picture box next to Photo.

   a. Change the shape of the box to a square.

   b. Click the right-arrow in the upper-right corner of the picture box.

      The **Picture Box Tasks** window opens.

   c. From the **Size Mode** dropdown list, choose **Zoom**.

d. To close the **Picture Box Tasks** window, click anywhere outside the window.

9. Build and run the project.

   a. From the Visual Studio **Build** menu, choose **Build Solution**.

   b. From the Visual Studio **Debug** menu, choose **Start Debugging**.

      The application connects to the SQL Anywhere sample database and displays the first row of the Products table in the text boxes and picture box.

   c. You can use the buttons on the control to scroll through the rows of the result set.

   d. You can go directly to a row in the result set by entering the row number in the scroll control.

   e. You can update values in the result set using the text boxes and save them by clicking the diskette icon.

   You have now created a simple, yet powerful, .NET application using Visual Studio, the Server Explorer, and the SQL Anywhere .NET Data Provider.

10. Shut down the application and then save your project.

Lesson 2. Add a synchronizing data control

This tutorial is a continuation of the tutorial described in "Lesson 1. Create a table viewer" on page 154. The complete application can be found in the ADO.NET project *samples-dir\SQLAnywhere\ADO.NET \SimpleViewer\SimpleViewer.sln*.

In this tutorial, you add a datagrid control to the form developed in the previous tutorial. This control updates automatically as you navigate through the result set.

To add a datagrid control

1. Start Visual Studio and load your MySimpleViewer project that you saved in the previous tutorial.
2. Right-click DataSet1 in the **Data Sources** window and choose **Edit DataSet With Designer**.

   The **DataSet Designer** window opens.
3. Right-click an empty area in the designer window and choose **Add** » **TableAdapter**.

   The **TableAdapter Configuration Wizard** appears.
   a. On the **Choose Your Data Connection** page, click **Next**.
   b. On the **Choose A Command Type** page, make sure **Use SQL Statements** is selected and then click **Next**.
   c. On the **Enter A SQL Statement** page, click **Query Builder**. The **Query Builder** and the **Add Table** windows appear.
   d. On the **Add Table** window, click the **Views** tab, select the ViewSalesOrders view, and click **Add**.
   e. Click **Close** to close the **Add Table** window.
4. Stretch the **Query Builder** window so that all sections of the window are clearly visible.
   a. Stretch the ViewSalesOrders window so that all of the checkboxes are visible.
   b. Select Region.
   c. Select Quantity.
   d. Select ProductID.
   e. In the grid below the ViewSalesOrders window, clear the checkbox under **Output** for the ProductID column. You want to use this column in your query, but you do not want to display it.
   f. For the ProductID column, type a question mark (?) in the **Filter** cell. This generates a WHERE clause for ProductID.

   A SQL query has been built that looks like the following:

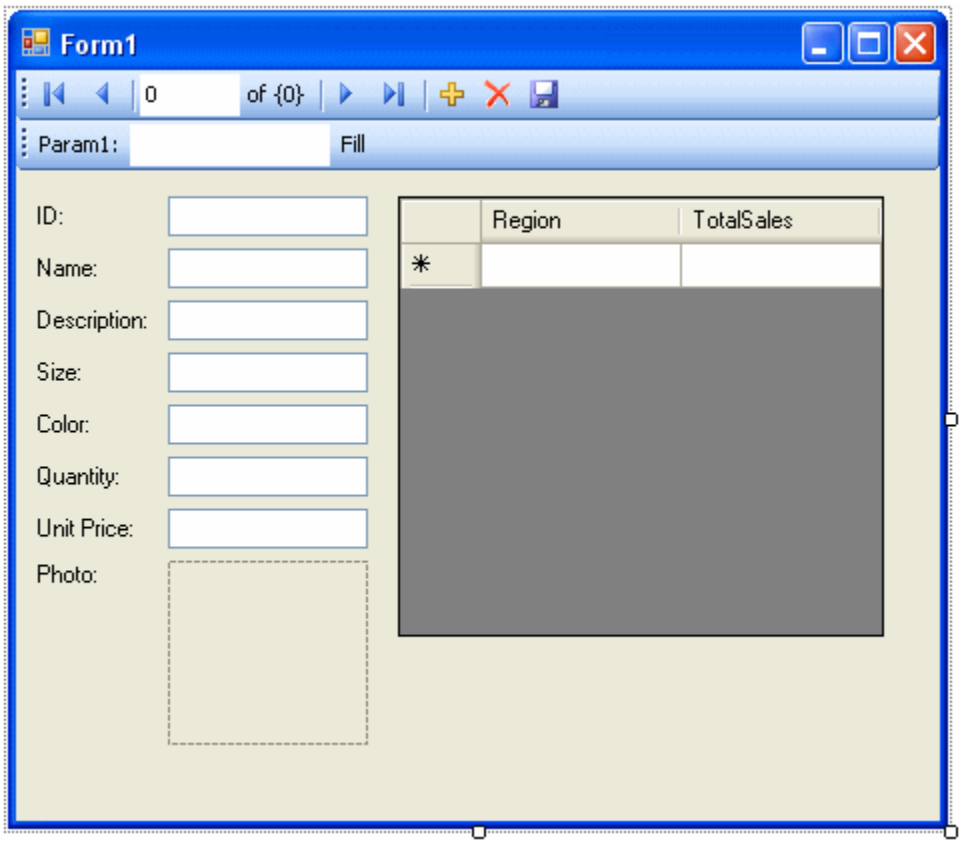
   ```
   SELECT   Region, Quantity
   FROM     GROUPO.ViewSalesOrders
   WHERE    (ProductID = :Param1)
   ```
5. Modify the SQL query as follows:
   a. Change `Quantity` to `SUM(Quantity) AS TotalSales`.
   b. Add `GROUP BY Region` to the end of the query following the WHERE clause.

   The modified SQL query now looks like this:


```
SELECT   Region, SUM(Quantity) as TotalSales
FROM     GROUPO.ViewSalesOrders
WHERE    (ProductID = :Param1)
GROUP BY Region
```

6. Click **OK** to close the **Query Builder** window.
7. Click **Finish** to close the **TableAdapter Configuration Wizard**.

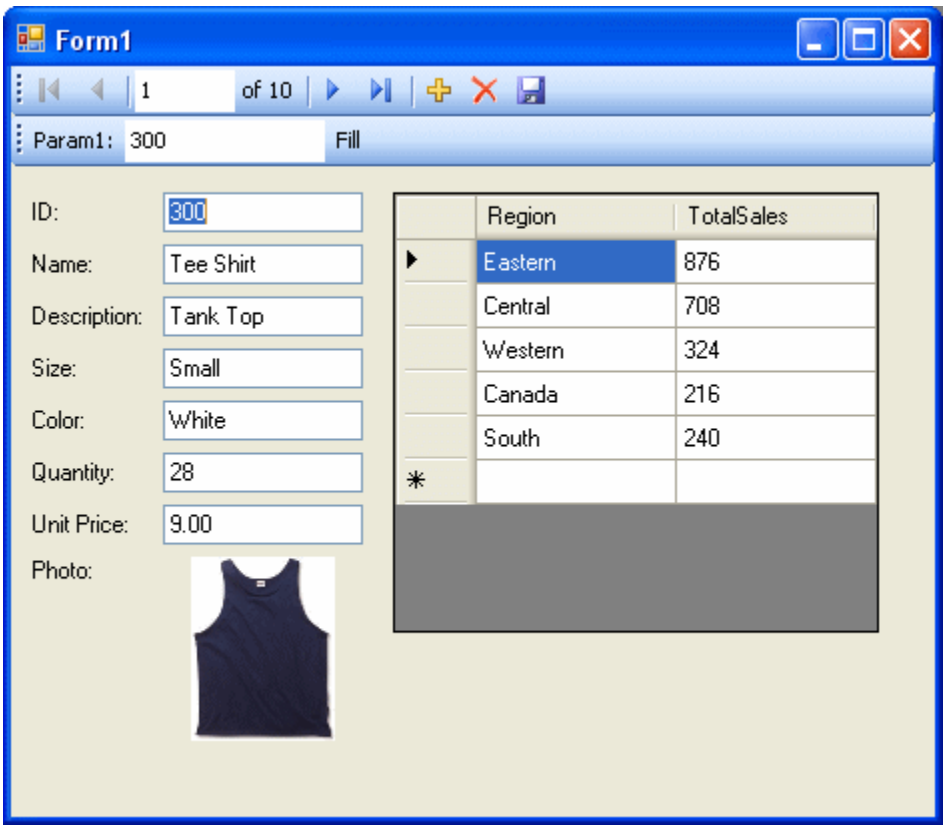
   A new **TableAdapter** called ViewSalesOrders has been added to the **DataSet Designer** window.
8. Click the form design tab (Form1).
   - Stretch the form to the right to make room for a new control.
9. Expand ViewSalesOrders in the **Data Sources** window.
   a. Click ViewSalesOrders and choose **DataGridView** from the dropdown list.
   b. Click ViewSalesOrders and drag it to your form (Form1).



   A datagrid view control appears on the form.
10. Build and run the project.
    - From the Visual Studio **Build** menu, choose **Build Solution**.
    - From the Visual Studio **Debug** menu, choose **Start Debugging**.

- In the **Param1** text box, enter a product ID number such as 300 and click **Fill**.

  The datagrid view displays a summary of sales by region for the product ID entered.



You can also use the other control on the form to move through the rows of the result set.

It would be ideal, however, if both controls could stay synchronized with each other. The next few steps show how to accomplish this.

11. Shut down the application and then save your project.
12. Delete the Fill strip on the form since you do not need it.
    - On the design form (Form1), right-click the Fill strip to the right of the word **Fill** and choose **Delete**.

      The Fill strip is removed from the form.
13. Synchronize the two controls as follows.
    a. On the design form (Form1), right-click the ID text box and choose **Properties**.
    b. Click the **Events** icon (it appears as a lightning bolt).
    c. Scroll down until you find the **TextChanged** event.
    d. Click **TextChanged** and choose **FillToolStripButton_Click** from the dropdown list. If you are using Visual Basic, the event is called **FillToolStripButton_Click**.

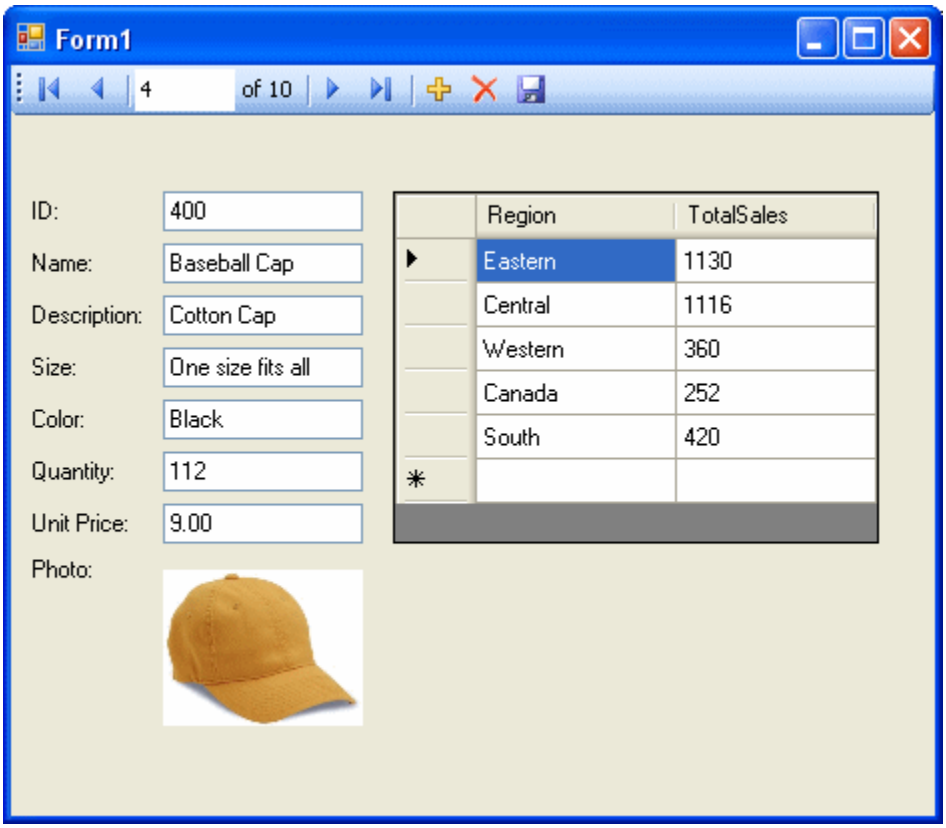
e. Double-click **FillToolStripButton_Click** and the form's code window opens on the `fillToolStripButton_Click` event handler.

f. Find the reference to `param1ToolStripTextBox` and change this to `iDTextBox`. If you are using Visual Basic, the text box is called `IDTextBox`.

g. Rebuild and run the project.

14. The application form now appears with a single navigation control.

- The datagrid view displays an updated summary of sales by region corresponding to the current product as you move through the result set.



You have now added a control that updates automatically as you navigate through the result set.

15. Shut down the application and then save your project.

In these tutorials, you saw how the powerful combination of Microsoft Visual Studio, the Server Explorer, and the SQL Anywhere .NET Data Provider can be used to create database applications.

CHAPTER 10

SQL Anywhere .NET 2.0 API Reference

Contents

iAnywhere.Data.SQLAnywhere namespace (.NET 2.0)

SABulkCopy class

Efficiently bulk load a SQL Anywhere table with data from another source. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SABulkCopy
  Implements IDisposable
```

C#

```
public sealed class SABulkCopy  : IDisposable
```

Remarks

Restrictions: The SABulkCopy class is not available in the .NET Compact Framework 2.0.

Implements: IDisposable

See also

- "SABulkCopy members" on page 164

SABulkCopy members

Public constructors

Member name	Description
SABulkCopy constructors	Initializes an SABulkCopy object.

Public properties

Member name	Description
BatchSize property	Gets or sets the number of rows in each batch. At the end of each batch, the rows in the batch are sent to the server.
BulkCopyTimeout property	Gets or sets the number of seconds for the operation to complete before it times out.
ColumnMappings property	Returns a collection of SABulkCopyColumnMapping items. Column mappings define the relationships between columns in the data source and columns in the destination.

Member name	Description
DestinationTableName property	Gets or sets the name of the destination table on the server.
NotifyAfter property	Gets or sets the number of rows to be processed before generating a notification event.

Public methods

Member name	Description
Close method	Closes the SABulkCopy instance.
Dispose method	Disposes of the SABulkCopy instance.
WriteToServer methods	Copies all rows in the supplied array of DataRow objects to a destination table specified by the DestinationTableName property of the SABulkCopy object.

Public events

Member name	Description
SARowsCopied event	This event occurs every time the number of rows specified by the NotifyAfter property have been processed.

See also

- "SABulkCopy class" on page 164

SABulkCopy constructors

Initializes an SABulkCopy object.

SABulkCopy(SAConnection) constructor

Syntax

Visual Basic

```
Public Sub New( _
   ByVal connection As SAConnection _
)
```

C#

```
public  SABulkCopy(
   SAConnection connection
);
```

Parameters

- **connection** The already open SAConnection that will be used to perform the bulk-copy operation. If the connection is not open, an exception is thrown in WriteToServer.

Remarks

Restrictions: The SABulkCopy class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164
- "SABulkCopy constructors" on page 165

SABulkCopy(String) constructor

Initializes an SABulkCopy object.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal connectionString As String _
)
```

C#

```
public  SABulkCopy(
  string  connectionString
);
```

Parameters

- **connectionString** The string defining the connection that will be opened for use by the SABulkCopy instance. A connection string is a semicolon-separated list of keyword=value pairs.

Remarks

This syntax opens a connection during WriteToServer using connectionString. The connection is closed at the end of WriteToServer.

Restrictions: The SABulkCopy class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164
- "SABulkCopy constructors" on page 165

SABulkCopy(String, SABulkCopyOptions) constructor

Initializes an SABulkCopy object.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal connectionString As String, _
  ByVal copyOptions As SABulkCopyOptions _
)
```

C#

```
public  SABulkCopy(
  string  connectionString,
  SABulkCopyOptions copyOptions
);
```

Parameters

- **connectionString** The string defining the connection that will be opened for use by the SABulkCopy instance. A connection string is a semicolon-separated list of keyword=value pairs.
- **copyOptions** A combination of values from the SABulkCopyOptions enumeration that determines which data source rows are copied to the destination table.

Remarks

This syntax opens a connection during WriteToServer using connectionString. The connection is closed at the end of WriteToServer. The copyOptions parameter has the effects described above.

Restrictions: The SABulkCopy class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164
- "SABulkCopy constructors" on page 165

SABulkCopy(SAConnection, SABulkCopyOptions, SATransaction) constructor

Initializes an SABulkCopy object.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal connection As SAConnection, _
  ByVal copyOptions As SABulkCopyOptions, _
  ByVal externalTransaction As SATransaction _
)
```

C#

```
public  SABulkCopy(
  SAConnection connection,
  SABulkCopyOptions copyOptions,
```

```
    SATransaction externalTransaction
);
```

Parameters

- **connection** The already open SAConnection that will be used to perform the bulk-copy operation. If the connection is not open, an exception is thrown in WriteToServer.
- **copyOptions** A combination of values from the SABulkCopyOptions enumeration that determines which data source rows are copied to the destination table.
- **externalTransaction** An existing SATransaction instance under which the bulk copy will occur. If externalTransaction is not NULL, then the bulk-copy operation is done within it. It is an error to specify both an external transaction and the UseInternalTransaction option.

Remarks

Restrictions: The SABulkCopy class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164
- "SABulkCopy constructors" on page 165

BatchSize property

Gets or sets the number of rows in each batch. At the end of each batch, the rows in the batch are sent to the server.

Syntax

Visual Basic

```
Public Property BatchSize As Integer
```

C#

```
public int BatchSize  { get; set; }
```

Property value

The number of rows in each batch. The default is 0.

Remarks

Setting this property to zero causes all the rows to be sent in one batch.

Setting this property to a value less than zero is an error.

If this value is changed while a batch is in progress, the current batch completes and any further batches use the new value.

See also

- "SABulkCopy class" on page 164

- "SABulkCopy members" on page 164

BulkCopyTimeout property

Gets or sets the number of seconds for the operation to complete before it times out.

Syntax

Visual Basic

Public Property **BulkCopyTimeout** As Integer

C#

public int **BulkCopyTimeout** { get; set; }

Property value

The default value is 30 seconds.

Remarks

A value of zero indicates no limit. This should be avoided because it may cause an indefinite wait.

If the operation times out, then all rows in the current transaction are rolled back and an SAException is raised.

Setting this property to a value less than zero is an error.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164

ColumnMappings property

Returns a collection of SABulkCopyColumnMapping items. Column mappings define the relationships between columns in the data source and columns in the destination.

Syntax

Visual Basic

Public Readonly Property **ColumnMappings** As SABulkCopyColumnMappingCollection

C#

public SABulkCopyColumnMappingCollection **ColumnMappings** { get;}

Property value

By default, it is an empty collection.

Remarks

The property cannot be modified while WriteToServer is executing.

If ColumnMappings is empty when WriteToServer is executed, then the first column in the source is mapped to the first column in the destination, the second to the second, and so on. This takes place as long as the column types are convertible, there are at least as many destination columns as source columns, and any extra destination columns are nullable.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164

DestinationTableName property

Gets or sets the name of the destination table on the server.

Syntax

Visual Basic

Public Property **DestinationTableName** As String

C#

public string **DestinationTableName** { get; set; }

Property value

The default value is a null reference. In Visual Basic it is Nothing.

Remarks

If the value is changed while WriteToServer is executing, the change has no effect.

If the value has not been set before a call to WriteToServer, an InvalidOperationException is raised.

It is an error to set the value to NULL or the empty string.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164

NotifyAfter property

Gets or sets the number of rows to be processed before generating a notification event.

Syntax

Visual Basic

Public Property **NotifyAfter** As Integer

C#

public int **NotifyAfter** { get; set; }

Property value

Zero is returned if the property has not been set.

Remarks

Changes made to NotifyAfter, while executing WriteToServer, do not take effect until after the next notification.

Setting this property to a value less than zero is an error.

The values of NotifyAfter and BulkCopyTimeOut are mutually exclusive, so the event can fire even if no rows have been sent to the database or committed.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164
- "BulkCopyTimeout property" on page 169

Close method

Closes the SABulkCopy instance.

Syntax

Visual Basic

Public Sub **Close()**

C#

public void **Close();**

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164

Dispose method

Disposes of the SABulkCopy instance.

Syntax

Visual Basic

NotOverridable Public Sub **Dispose()**

C#

```
public void Dispose();
```

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164

WriteToServer methods

Copies all rows in the supplied array of DataRow objects to a destination table specified by the DestinationTableName property of the SABulkCopy object.

WriteToServer(DataRow[]) method

Copies all rows in the supplied array of DataRow objects to a destination table specified by the DestinationTableName property of the SABulkCopy object.

Syntax

Visual Basic

```
Public Sub WriteToServer( _
  ByVal rows As DataRow() _
)
```

C#

```
public void WriteToServer(
  DataRow[] rows
);
```

Parameters

- **rows** An array of System.Data.DataRow objects that will be copied to the destination table.

Remarks

Restrictions: The SABulkCopy class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164
- "WriteToServer methods" on page 172
- "DestinationTableName property" on page 170

WriteToServer(DataTable) method

Copies all rows in the supplied DataTable to a destination table specified by the DestinationTableName property of the SABulkCopy object.

Syntax

Visual Basic

```
Public Sub WriteToServer( _
  ByVal table As DataTable _
)
```

C#

```
public void WriteToServer(
  DataTable table
);
```

Parameters

- **table** A System.Data.DataTable whose rows will be copied to the destination table.

Remarks

Restrictions: The SABulkCopy class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164
- "WriteToServer methods" on page 172
- "DestinationTableName property" on page 170

WriteToServer(IDataReader) method

Copies all rows in the supplied IDataReader to a destination table specified by the DestinationTableName property of the SABulkCopy object.

Syntax

Visual Basic

```
Public Sub WriteToServer( _
  ByVal reader As IDataReader _
)
```

C#

```
public void WriteToServer(
  IDataReader reader
);
```

Parameters

- **reader** A System.Data.IDataReader whose rows will be copied to the destination table.

Remarks

Restrictions: The SABulkCopy class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164
- "WriteToServer methods" on page 172
- "DestinationTableName property" on page 170

WriteToServer(DataTable, DataRowState) method

Copies all rows in the supplied DataTable with the specified row state to a destination table specified by the DestinationTableName property of the SABulkCopy object.

Syntax

Visual Basic

```
Public Sub WriteToServer( _
  ByVal table As DataTable, _
  ByVal rowState As DataRowState _
)
```

C#

```
public void WriteToServer(
  DataTable table,
  DataRowState rowState
);
```

Parameters

- **table** A System.Data.DataTable whose rows will be copied to the destination table.
- **rowState** A value from the System.Data.DataRowState enumeration. Only rows matching the row state are copied to the destination.

Remarks

Only those rows matching the row state are copied.

Restrictions: The SABulkCopy class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164
- "WriteToServer methods" on page 172
- "DestinationTableName property" on page 170

SARowsCopied event

This event occurs every time the number of rows specified by the NotifyAfter property have been processed.

Syntax

Visual Basic

Public Event **SARowsCopied** As SARowsCopiedEventHandler

C#

public event SARowsCopiedEventHandler **SARowsCopied** ;

Remarks

The receipt of an SARowsCopied event does not imply that any rows have been sent to the database server or committed. You cannot call the Close method from this event.

See also

- "SABulkCopy class" on page 164
- "SABulkCopy members" on page 164
- "NotifyAfter property" on page 170

SABulkCopyColumnMapping class

Defines the mapping between a column in an SABulkCopy instance's data source and a column in the instance's destination table. This class cannot be inherited.

Syntax

Visual Basic

Public NotInheritable Class **SABulkCopyColumnMapping**

C#

public sealed class **SABulkCopyColumnMapping**

Remarks

Restrictions: The SABulkCopyColumnMapping class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMapping members" on page 176

SABulkCopyColumnMapping members

Public constructors

Member name	Description
SABulkCopyColumnMapping constructors	Initializes a new instance of the "SABulkCopyColumnMapping class" on page 175.

Public properties

Member name	Description
DestinationColumn property	Gets or sets the name of the column in the destination database table being mapped to.
DestinationOrdinal property	Gets or sets the ordinal value of the column in the destination table being mapped to.
SourceColumn property	Gets or sets the name of the column being mapped in the data source.
SourceOrdinal property	Gets or sets ordinal position of the source column within the data source.

See also

- "SABulkCopyColumnMapping class" on page 175

SABulkCopyColumnMapping constructors

Initializes a new instance of the "SABulkCopyColumnMapping class" on page 175.

SABulkCopyColumnMapping() constructor

Creates a new column mapping, using column ordinals or names to refer to source and destination columns.

Syntax

Visual Basic

Public Sub **New()**

C#

public **SABulkCopyColumnMapping();**

Remarks

Restrictions: The SABulkCopyColumnMapping class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMapping class" on page 175
- "SABulkCopyColumnMapping members" on page 176
- "SABulkCopyColumnMapping constructors" on page 176

SABulkCopyColumnMapping(Int32, Int32) constructor

Creates a new column mapping, using column ordinals to refer to source and destination columns.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal sourceColumnOrdinal As Integer, _
  ByVal destinationColumnOrdinal As Integer _
)
```

C#

```
public  SABulkCopyColumnMapping(
  int sourceColumnOrdinal,
  int destinationColumnOrdinal
);
```

Parameters

- **sourceColumnOrdinal** The ordinal position of the source column within the data source. The first column in a data source has ordinal position zero.
- **destinationColumnOrdinal** The ordinal position of the destination column within the destination table. The first column in a table has ordinal position zero.

Remarks

Restrictions: The SABulkCopyColumnMapping class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMapping class" on page 175
- "SABulkCopyColumnMapping members" on page 176
- "SABulkCopyColumnMapping constructors" on page 176

SABulkCopyColumnMapping(Int32, String) constructor

Creates a new column mapping, using a column ordinal to refer to the source column and a column name to refer to the destination column.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal sourceColumnOrdinal As Integer, _
  ByVal destinationColumn As String _
)
```

C#

```
public  SABulkCopyColumnMapping(
  int sourceColumnOrdinal,
  string  destinationColumn
);
```

Parameters

- **sourceColumnOrdinal** The ordinal position of the source column within the data source. The first column in a data source has ordinal position zero.
- **destinationColumn** The name of the destination column within the destination table.

Remarks

Restrictions: The SABulkCopyColumnMapping class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMapping class" on page 175
- "SABulkCopyColumnMapping members" on page 176
- "SABulkCopyColumnMapping constructors" on page 176

SABulkCopyColumnMapping(String, Int32) constructor

Creates a new column mapping, using a column name to refer to the source column and a column ordinal to refer to the destination column.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal sourceColumn As String, _
  ByVal destinationColumnOrdinal As Integer _
)
```

C#


```
public  SABulkCopyColumnMapping(
  string  sourceColumn,
  int destinationColumnOrdinal
);
```

Parameters

- **sourceColumn** The name of the source column within the data source.
- **destinationColumnOrdinal** The ordinal position of the destination column within the destination table. The first column in a table has ordinal position zero.

Remarks

Restrictions: The SABulkCopyColumnMapping class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMapping class" on page 175
- "SABulkCopyColumnMapping members" on page 176
- "SABulkCopyColumnMapping constructors" on page 176

SABulkCopyColumnMapping(String, String) constructor

Creates a new column mapping, using column names to refer to source and destination columns.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal sourceColumn As String, _
  ByVal destinationColumn As String _
)
```

C#

```
public  SABulkCopyColumnMapping(
  string  sourceColumn,
  string  destinationColumn
);
```

Parameters

- **sourceColumn** The name of the source column within the data source.
- **destinationColumn** The name of the destination column within the destination table.

Remarks

Restrictions: The SABulkCopyColumnMapping class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMapping class" on page 175

- "SABulkCopyColumnMapping members" on page 176
- "SABulkCopyColumnMapping constructors" on page 176

DestinationColumn property

Gets or sets the name of the column in the destination database table being mapped to.

Syntax

Visual Basic

Public Property **DestinationColumn** As String

C#

public string **DestinationColumn** { get; set; }

Property value

A string specifying the name of the column in the destination table or a null reference (Nothing in Visual Basic) if the DestinationOrdinal property has priority.

Remarks

The DestinationColumn property and DestinationOrdinal property are mutually exclusive. The most recently set value takes priority.

Setting the DestinationColumn property causes the DestinationOrdinal property to be set to -1. Setting the DestinationOrdinal property causes the DestinationColumn property to be set to a null reference (Nothing in Visual Basic).

It is an error to set DestinationColumn to null or the empty string.

See also

- "SABulkCopyColumnMapping class" on page 175
- "SABulkCopyColumnMapping members" on page 176
- "DestinationOrdinal property" on page 180

DestinationOrdinal property

Gets or sets the ordinal value of the column in the destination table being mapped to.

Syntax

Visual Basic

Public Property **DestinationOrdinal** As Integer

C#

public int **DestinationOrdinal** { get; set; }

Property value

An integer specifying the ordinal of the column being mapped to in the destination table or -1 if the property is not set.

Remarks

The DestinationColumn property and DestinationOrdinal property are mutually exclusive. The most recently set value takes priority.

Setting the DestinationColumn property causes the DestinationOrdinal property to be set to -1. Setting the DestinationOrdinal property causes the DestinationColumn property to be set to a null reference (Nothing in Visual Basic).

See also

- "SABulkCopyColumnMapping class" on page 175
- "SABulkCopyColumnMapping members" on page 176
- "DestinationColumn property" on page 180

SourceColumn property

Gets or sets the name of the column being mapped in the data source.

Syntax

Visual Basic

```
Public Property SourceColumn As String
```

C#

```
public string  SourceColumn  { get; set; }
```

Property value

A string specifying the name of the column in the data source or a null reference (Nothing in Visual Basic) if the SourceOrdinal property has priority.

Remarks

The SourceColumn property and SourceOrdinal property are mutually exclusive. The most recently set value takes priority.

Setting the SourceColumn property causes the SourceOrdinal property to be set to -1. Setting the SourceOrdinal property causes the SourceColumn property to be set to a null reference (Nothing in Visual Basic).

It is an error to set SourceColumn to null or the empty string.

See also

- "SABulkCopyColumnMapping class" on page 175
- "SABulkCopyColumnMapping members" on page 176
- "SourceOrdinal property" on page 182

SourceOrdinal property

Gets or sets ordinal position of the source column within the data source.

Syntax

Visual Basic

```
Public Property SourceOrdinal As Integer
```

C#

```
public int SourceOrdinal  { get; set; }
```

Property value

An integer specifying the ordinal of the column in the data source or -1 if the property is not set.

Remarks

The SourceColumn property and SourceOrdinal property are mutually exclusive. The most recently set value takes priority.

Setting the SourceColumn property causes the SourceOrdinal property to be set to -1. Setting the SourceOrdinal property causes the SourceColumn property to be set to a null reference (Nothing in Visual Basic).

See also

- "SABulkCopyColumnMapping class" on page 175
- "SABulkCopyColumnMapping members" on page 176
- "SourceColumn property" on page 181

SABulkCopyColumnMappingCollection class

A collection of SABulkCopyColumnMapping objects that inherits from System.Collections.CollectionBase. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SABulkCopyColumnMappingCollection
  Inherits CollectionBase
```

C#

```
public sealed class SABulkCopyColumnMappingCollection : CollectionBase
```

Remarks

Restrictions: The SABulkCopyColumnMappingCollection class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMappingCollection members" on page 183

SABulkCopyColumnMappingCollection members

Public properties

Member name	Description
Capacity (inherited from CollectionBase)	
Count (inherited from CollectionBase)	
Item property	Gets the SABulkCopyColumnMapping object at the specified index.

Public methods

Member name	Description
Add methods	Adds the specified SABulkCopyColumnMapping object to the collection.
Clear (inherited from CollectionBase)	
Contains method	Gets a value indicating whether a specified SABulkCopyColumnMapping object exists in the collection.
CopyTo method	Copies the elements of the SABulkCopyColumnMappingCollection to an array of SABulkCopyColumnMapping items, starting at a particular index.
GetEnumerator (inherited from CollectionBase)	
IndexOf method	Gets or sets the index of the specified SABulkCopyColumnMapping object within the collection.
Remove method	Removes the specified SABulkCopyColumnMapping element from the SABulkCopyColumnMappingCollection.
RemoveAt method	Removes the mapping at the specified index from the collection.

See also

- "SABulkCopyColumnMappingCollection class" on page 182

Item property

Gets the SABulkCopyColumnMapping object at the specified index.

Syntax

Visual Basic

```
Public Readonly Property Item ( _
  ByVal index As Integer _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping this [
  int index
] { get;}
```

Parameters

- **index** The zero-based index of the SABulkCopyColumnMapping object to find.

Property value

An SABulkCopyColumnMapping object is returned.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183

Add methods

Adds the specified SABulkCopyColumnMapping object to the collection.

Add(SABulkCopyColumnMapping) method

Adds the specified SABulkCopyColumnMapping object to the collection.

Syntax

Visual Basic

```
Public Function Add( _
  ByVal bulkCopyColumnMapping As SABulkCopyColumnMapping _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping Add(
  SABulkCopyColumnMapping bulkCopyColumnMapping
);
```


Parameters

- **bulkCopyColumnMapping** The SABulkCopyColumnMapping object that describes the mapping to be added to the collection.

Remarks

Restrictions: The SABulkCopyColumnMappingCollection class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183
- "Add methods" on page 184
- "SABulkCopyColumnMapping class" on page 175

Add(Int32, Int32) method

Creates a new SABulkCopyColumnMapping object using ordinals to specify both source and destination columns, and adds the mapping to the collection.

Syntax

Visual Basic

```
Public Function Add( _
   ByVal sourceColumnOrdinal As Integer, _
   ByVal destinationColumnOrdinal As Integer _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping Add(
   int sourceColumnOrdinal,
   int destinationColumnOrdinal
);
```

Parameters

- **sourceColumnOrdinal** The ordinal position of the source column within the data source.
- **destinationColumnOrdinal** The ordinal position of the destination column within the destination table.

Remarks

Restrictions: The SABulkCopyColumnMappingCollection class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183
- "Add methods" on page 184

Add(Int32, String) method

Creates a new SABulkCopyColumnMapping object using a column ordinal to refer to the source column and a column name to refer to the destination column, and adds mapping to the collection.

Syntax

Visual Basic

```
Public Function Add( _
   ByVal sourceColumnOrdinal As Integer, _
   ByVal destinationColumn As String _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping Add(
   int sourceColumnOrdinal,
   string  destinationColumn
);
```

Parameters

- **sourceColumnOrdinal** The ordinal position of the source column within the data source.
- **destinationColumn** The name of the destination column within the destination table.

Remarks

Restrictions: The SABulkCopyColumnMappingCollection class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183
- "Add methods" on page 184

Add(String, Int32) method

Creates a new SABulkCopyColumnMapping object using a column name to refer to the source column and a column ordinal to refer to the destination the column, and adds the mapping to the collection.

Creates a new column mapping, using column ordinals or names to refer to source and destination columns.

Syntax

Visual Basic

```
Public Function Add( _
   ByVal sourceColumn As String, _
   ByVal destinationColumnOrdinal As Integer _
) As SABulkCopyColumnMapping
```

C#


```
public SABulkCopyColumnMapping Add(
  string  sourceColumn,
  int destinationColumnOrdinal
);
```

Parameters

- **sourceColumn** The name of the source column within the data source.
- **destinationColumnOrdinal** The ordinal position of the destination column within the destination table.

Remarks

Restrictions: The SABulkCopyColumnMappingCollection class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183
- "Add methods" on page 184

Add(String, String) method

Creates a new SABulkCopyColumnMapping object using column names to specify both source and destination columns, and adds the mapping to the collection.

Syntax

Visual Basic

```
Public Function Add( _
  ByVal sourceColumn As String, _
  ByVal destinationColumn As String _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping Add(
  string  sourceColumn,
  string  destinationColumn
);
```

Parameters

- **sourceColumn** The name of the source column within the data source.
- **destinationColumn** The name of the destination column within the destination table.

Remarks

Restrictions: The SABulkCopyColumnMappingCollection class is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183
- "Add methods" on page 184

Contains method

Gets a value indicating whether a specified SABulkCopyColumnMapping object exists in the collection.

Syntax

Visual Basic

```
Public Function Contains( _
  ByVal value As SABulkCopyColumnMapping _
) As Boolean
```

C#

```
public bool Contains(
  SABulkCopyColumnMapping value
);
```

Parameters

- **value** A valid SABulkCopyColumnMapping object.

Return value

True if the specified mapping exists in the collection; otherwise, false.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183

CopyTo method

Copies the elements of the SABulkCopyColumnMappingCollection to an array of SABulkCopyColumnMapping items, starting at a particular index.

Syntax

Visual Basic

```
Public Sub CopyTo( _
  ByVal array As SABulkCopyColumnMapping(), _
  ByVal index As Integer _
)
```

C#

```
public void CopyTo(
```



```
  SABulkCopyColumnMapping[] array,
  int index
);
```

Parameters

- **array** The one-dimensional SABulkCopyColumnMapping array that is the destination of the elements copied from SABulkCopyColumnMappingCollection. The array must have zero-based indexing.
- **index** The zero-based index in the array at which copying begins.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183

IndexOf method

Gets or sets the index of the specified SABulkCopyColumnMapping object within the collection.

Syntax

Visual Basic

```
Public Function IndexOf( _
  ByVal value As SABulkCopyColumnMapping _
) As Integer
```

C#

```
public int IndexOf(
  SABulkCopyColumnMapping value
);
```

Parameters

- **value** The SABulkCopyColumnMapping object to search for.

Return value

The zero-based index of the column mapping is returned, or -1 is returned if the column mapping is not found in the collection.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183

Remove method

Removes the specified SABulkCopyColumnMapping element from the SABulkCopyColumnMappingCollection.

Syntax

Visual Basic

```
Public Sub Remove( _
   ByVal value As SABulkCopyColumnMapping _
)
```

C#

```
public void Remove(
   SABulkCopyColumnMapping value
);
```

Parameters

- **value** The SABulkCopyColumnMapping object to be removed from the collection.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183

RemoveAt method

Removes the mapping at the specified index from the collection.

Syntax

Visual Basic

```
Public Sub RemoveAt( _
   ByVal index As Integer _
)
```

C#

```
public void RemoveAt(
   int index
);
```

Parameters

- **index** The zero-based index of the SABulkCopyColumnMapping object to be removed from the collection.

See also

- "SABulkCopyColumnMappingCollection class" on page 182
- "SABulkCopyColumnMappingCollection members" on page 183

SABulkCopyOptions enumeration

A bitwise flag that specifies one or more options to use with an instance of SABulkCopy.

Syntax

Visual Basic

Public Enum **SABulkCopyOptions**

C#

public enum **SABulkCopyOptions**

Remarks

The SABulkCopyOptions enumeration is used when you construct an SABulkCopy object to specify how the WriteToServer methods will behave.

Restrictions: The SABulkCopyOptions class is not available in the .NET Compact Framework 2.0.

The CheckConstraints and KeepNulls options are not supported.

Members

Member name	Description	Value
Default	Specifying only this value causes the default behavior to be used. By default, triggers are enabled.	0
DoNotFireTriggers	When specified, triggers are not fired. Disabling triggers requires DBA permission. Triggers are disabled for the connection at the start of WriteToServer and the value is restored at the end of the method.	1
KeepIdentity	When specified, the source values to be copied into an identity column are preserved. By default, new identity values are generated in the destination table.	2
TableLock	When specified the table is locked using the command LOCK TABLE table_name WITH HOLD IN SHARE MODE. This lock is in place until the connection is closed.	4
UseInternalTransaction	When specified, each batch of the bulk-copy operation is executed within a transaction. When not specified, transaction aren't used. If you indicate this option and also provide an SATransaction object to the constructor, a System.ArgumentException occurs.	8

See also

- "SABulkCopy class" on page 164

SACommand class

A SQL statement or stored procedure that is executed against a SQL Anywhere database. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SACommand
  Inherits DbCommand
  Implements ICloneable
```

C#

```
public sealed class SACommand : DbCommand,
  ICloneable
```

Remarks

Implements:ICloneable

For more information, see "Accessing and manipulating data" on page 115.

See also

- "SACommand members" on page 192

SACommand members

Public constructors

Member name	Description
SACommand constructors	Initializes a new instance of the "SACommand class" on page 192.

Public properties

Member name	Description
CommandText property	Gets or sets the text of a SQL statement or stored procedure.
CommandTimeout property	Gets or sets the wait time in seconds before terminating an attempt to execute a command and generating an error.
CommandType property	Gets or sets the type of command represented by an SACommand.
Connection property	Gets or sets the connection object to which the SACommand object applies.

Member name	Description
DesignTimeVisible property	Gets or sets a value that indicates if the SACommand should be visible in a Windows Form Designer control. The default is true.
Parameters property	A collection of parameters for the current statement. Use question marks in the CommandText to indicate parameters.
Transaction property	Specifies the SATransaction object in which the SACommand executes.
UpdatedRowSource property	Gets or sets how command results are applied to the DataRow when used by the Update method of the SADataAdapter.

Public methods

Member name	Description
BeginExecuteNonQuery methods	Initiates the asynchronous execution of a SQL statement or stored procedure that is described by this SACommand.
BeginExecuteReader methods	Initiates the asynchronous execution of a SQL statement or stored procedure that is described by this SACommand, and retrieves one or more result sets from the database server.
Cancel method	Cancels the execution of an SACommand object.
CreateParameter method	Provides an SAParameter object for supplying parameters to SACommand objects.
EndExecuteNonQuery method	Finishes asynchronous execution of a SQL statement or stored procedure.
EndExecuteReader method	Finishes asynchronous execution of a SQL statement or stored procedure, returning the requested SADataReader.
ExecuteNonQuery method	Executes a statement that does not return a result set, such as an INSERT, UPDATE, DELETE, or data definition statement.
ExecuteReader methods	Executes a SQL statement that returns a result set.
ExecuteScalar method	Executes a statement that returns a single value. If this method is called on a query that returns multiple rows and columns, only the first column of the first row is returned.
Prepare method	Prepares or compiles the SACommand on the data source.
ResetCommandTimeout method	Resets the CommandTimeout property to its default value of 30 seconds.

See also

- "SACommand class" on page 192

SACommand constructors

Initializes a new instance of the "SACommand class" on page 192.

SACommand() constructor

Initializes an SACommand object.

Syntax

Visual Basic

```
Public Sub New()
```

C#

```
public  SACommand();
```

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "SACommand constructors" on page 194

SACommand(String) constructor

Initializes an SACommand object.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal cmdText As String _
)
```

C#

```
public  SACommand(
  string  cmdText
);
```

Parameters

- **cmdText** The text of the SQL statement or stored procedure. For parameterized statements, use a question mark (?) placeholder to pass parameters.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "SACommand constructors" on page 194

SACommand(String, SAConnection) constructor

A SQL statement or stored procedure that is executed against a SQL Anywhere database.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal cmdText As String, _
  ByVal connection As SAConnection _
)
```

C#

```
public  SACommand(
  string  cmdText,
  SAConnection connection
);
```

Parameters

- **cmdText** The text of the SQL statement or stored procedure. For parameterized statements, use a question mark (?) placeholder to pass parameters.
- **connection** The current connection.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "SACommand constructors" on page 194

SACommand(String, SAConnection, SATransaction) constructor

A SQL statement or stored procedure that is executed against a SQL Anywhere database.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal cmdText As String, _
  ByVal connection As SAConnection, _
  ByVal transaction As SATransaction _
)
```


C#

```
public  SACommand(
  string  cmdText,
  SAConnection connection,
  SATransaction transaction
);
```

Parameters

- **cmdText** The text of the SQL statement or stored procedure. For parameterized statements, use a question mark (?) placeholder to pass parameters.
- **connection** The current connection.
- **transaction** The SATransaction object in which the SAConnection executes.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "SACommand constructors" on page 194
- "SATransaction class" on page 416

CommandText property

Gets or sets the text of a SQL statement or stored procedure.

Syntax

Visual Basic

```
Public Overrides Property CommandText As String
```

C#

```
public override string  CommandText  { get; set; }
```

Property value

The SQL statement or the name of the stored procedure to execute. The default is an empty string.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "SACommand() constructor" on page 194

CommandTimeout property

Gets or sets the wait time in seconds before terminating an attempt to execute a command and generating an error.

Syntax

Visual Basic

Public Overrides Property **CommandTimeout** As Integer

C#

public override int **CommandTimeout** { get; set; }

Property value

The default value is 30 seconds.

Remarks

A value of 0 indicates no limit. This should be avoided because it may cause the attempt to execute a command to wait indefinitely.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192

CommandType property

Gets or sets the type of command represented by an SACommand.

Syntax

Visual Basic

Public Overrides Property **CommandType** As CommandType

C#

public override CommandType **CommandType** { get; set; }

Property value

One of the CommandType values. The default is CommandType.Text.

Remarks

Supported command types are as follows:

- CommandType.StoredProcedure When you specify this CommandType, the command text must be the name of a stored procedure and you must supply any arguments as SAParameter objects.
- CommandType.Text This is the default value.

When the CommandType property is set to StoredProcedure, the CommandText property should be set to the name of the stored procedure. The command executes this stored procedure when you call one of the Execute methods.

Use a question mark (?) placeholder to pass parameters. For example:

```
SELECT * FROM Customers WHERE ID = ?
```

The order in which SAParameter objects are added to the SAParameterCollection must directly correspond to the position of the question mark placeholder for the parameter.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192

Connection property

Gets or sets the connection object to which the SACommand object applies.

Syntax

Visual Basic

Public Property **Connection** As SAConnection

C#

public SAConnection **Connection** { get; set; }

Property value

The default value is a null reference. In Visual Basic it is Nothing.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192

DesignTimeVisible property

Gets or sets a value that indicates if the SACommand should be visible in a Windows Form Designer control. The default is true.

Syntax

Visual Basic

Public Overrides Property **DesignTimeVisible** As Boolean

C#

public override bool **DesignTimeVisible** { get; set; }

Property value

True if this SACommand instance should be visible, false if this instance should not be visible. The default is false.

See also

- "SACommand class" on page 192

- "SACommand members" on page 192

Parameters property

A collection of parameters for the current statement. Use question marks in the CommandText to indicate parameters.

Syntax

Visual Basic

Public Readonly Property **Parameters** As SAParameterCollection

C#

public SAParameterCollection **Parameters** { get;}

Property value

The parameters of the SQL statement or stored procedure. The default value is an empty collection.

Remarks

When CommandType is set to Text, pass parameters using the question mark placeholder. For example:

```
SELECT * FROM Customers WHERE ID = ?
```

The order in which SAParameter objects are added to the SAParameterCollection must directly correspond to the position of the question mark placeholder for the parameter in the command text.

When the parameters in the collection do not match the requirements of the query to be executed, an error may result or an exception may be thrown.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "SAParameterCollection class" on page 373

Transaction property

Specifies the SATransaction object in which the SACommand executes.

Syntax

Visual Basic

Public Property **Transaction** As SATransaction

C#

public SATransaction **Transaction** { get; set; }

Property value

The default value is a null reference. In Visual Basic, this is Nothing.

Remarks

You cannot set the Transaction property if it is already set to a specific value and the command is executing. If you set the transaction property to an SATransaction object that is not connected to the same SAConnection object as the SACommand object, an exception will be thrown the next time you attempt to execute a statement.

For more information, see "Transaction processing" on page 134.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "SATransaction class" on page 416

UpdatedRowSource property

Gets or sets how command results are applied to the DataRow when used by the Update method of the SADataAdapter.

Syntax

Visual Basic

```
Public Overrides Property UpdatedRowSource As UpdateRowSource
```

C#

```
public override UpdateRowSource UpdatedRowSource  { get; set; }
```

Property value

One of the UpdatedRowSource values. The default value is UpdateRowSource.OutputParameters. If the command is automatically generated, this property is UpdateRowSource.None.

Remarks

UpdatedRowSource.Both, which returns both resultset and output parameters, is not supported.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192

BeginExecuteNonQuery methods

Initiates the asynchronous execution of a SQL statement or stored procedure that is described by this SACommand.

BeginExecuteNonQuery() method

Initiates the asynchronous execution of a SQL statement or stored procedure that is described by this SACommand.

Syntax

Visual Basic

```
Public Function BeginExecuteNonQuery() As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteNonQuery();
```

Return value

An IAsyncResult that can be used to poll, wait for results, or both; this value is also needed when invoking EndExecuteNonQuery(IAsyncResult), which returns the number of affected rows.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "BeginExecuteNonQuery methods" on page 200
- "EndExecuteNonQuery method" on page 206

BeginExecuteNonQuery(AsyncCallback, Object) method

Initiates the asynchronous execution of a SQL statement or stored procedure that is described by this SACommand, given a callback procedure and state information.

Syntax

Visual Basic

```
Public Function BeginExecuteNonQuery( _
   ByVal callback As AsyncCallback, _
   ByVal stateObject As Object _
) As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteNonQuery(
   AsyncCallback callback,
   object stateObject
);
```

Parameters

- **callback** An AsyncCallback delegate that is invoked when the command's execution has completed. Pass null (Nothing in Microsoft Visual Basic) to indicate that no callback is required.
- **stateObject** A user-defined state object that is passed to the callback procedure. Retrieve this object from within the callback procedure using the IAsyncResult.AsyncState.

Return value

An IAsyncResult that can be used to poll, wait for results, or both; this value is also needed when invoking EndExecuteNonQuery(IAsyncResult), which returns the number of affected rows.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "BeginExecuteNonQuery methods" on page 200
- "EndExecuteNonQuery method" on page 206

BeginExecuteReader methods

Initiates the asynchronous execution of a SQL statement or stored procedure that is described by this SACommand, and retrieves one or more result sets from the database server.

BeginExecuteReader() method

Initiates the asynchronous execution of a SQL statement or stored procedure that is described by this SACommand, and retrieves one or more result sets from the database server.

Syntax

Visual Basic

Public Function **BeginExecuteReader()** As IAsyncResult

C#

public IAsyncResult **BeginExecuteReader();**

Return value

An IAsyncResult that can be used to poll, wait for results, or both; this value is also needed when invoking EndExecuteReader(IAsyncResult), which returns an SADataReader object that can be used to retrieve the returned rows.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "BeginExecuteReader methods" on page 202
- "EndExecuteReader method" on page 208
- "SADataReader class" on page 289

BeginExecuteReader(CommandBehavior) method

Initiates the asynchronous execution of a SQL statement or stored procedure that is described by this SACommand, and retrieves one or more result sets from the server.

Syntax

Visual Basic

```
Public Function BeginExecuteReader( _
  ByVal behavior As CommandBehavior _
) As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteReader(
  CommandBehavior behavior
);
```

Parameters

- **behavior** A bitwise combination of CommandBehavior flags describing the results of the query and its effect on the connection.

Return value

An IAsyncResult that can be used to poll, wait for results, or both; this value is also needed when invoking EndExecuteReader(IAsyncResult), which returns an SADataReader object that can be used to retrieve the returned rows.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "BeginExecuteReader methods" on page 202
- "EndExecuteReader method" on page 208
- "SADataReader class" on page 289

BeginExecuteReader(AsyncCallback, Object) method

Initiates the asynchronous execution of a SQL statement that is described by this SA Command, and retrieves the result set, given a callback procedure and state information.

Syntax

Visual Basic

```
Public Function BeginExecuteReader( _
  ByVal callback As AsyncCallback, _
  ByVal stateObject As Object _
) As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteReader(
  AsyncCallback callback,
  object stateObject
);
```

Parameters

- **callback** An AsyncCallback delegate that is invoked when the command's execution has completed. Pass null (Nothing in Microsoft Visual Basic) to indicate that no callback is required.
- **stateObject** A user-defined state object that is passed to the callback procedure. Retrieve this object from within the callback procedure using the IAsyncResult.AsyncState.

Return value

An IAsyncResult that can be used to poll, wait for results, or both; this value is also needed when invoking EndExecuteReader(IAsyncResult), which returns an SADataReader object that can be used to retrieve the returned rows.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "BeginExecuteReader methods" on page 202
- "EndExecuteReader method" on page 208
- "SADataReader class" on page 289

BeginExecuteReader(AsyncCallback, Object, CommandBehavior) method

Initiates the asynchronous execution of a SQL statement or stored procedure that is described by this SACommand, and retrieves one or more result sets from the server.

Syntax

Visual Basic

```
Public Function BeginExecuteReader( _
   ByVal callback As AsyncCallback, _
   ByVal stateObject As Object, _
   ByVal behavior As CommandBehavior _
) As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteReader(
   AsyncCallback callback,
   object stateObject,
   CommandBehavior behavior
);
```

Parameters

- **callback** An AsyncCallback delegate that is invoked when the command's execution has completed. Pass null (Nothing in Microsoft Visual Basic) to indicate that no callback is required.
- **stateObject** A user-defined state object that is passed to the callback procedure. Retrieve this object from within the callback procedure using the IAsyncResult.AsyncState.
- **behavior** A bitwise combination of CommandBehavior flags describing the results of the query and its effect on the connection.

Return value

An IAsyncResult that can be used to poll, wait for results, or both; this value is also needed when invoking EndExecuteReader(IAsyncResult), which returns an SADataReader object that can be used to retrieve the returned rows.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "BeginExecuteReader methods" on page 202
- "EndExecuteReader method" on page 208
- "SADataReader class" on page 289

Cancel method

Cancels the execution of an SACommand object.

Syntax

Visual Basic

Public Overrides Sub **Cancel()**

C#

public override void **Cancel();**

Remarks

If there is nothing to cancel, nothing happens. If there is a command in process and the attempt to cancel fails, not exception is generated.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192

CreateParameter method

Provides an SAParameter object for supplying parameters to SACommand objects.

Syntax

Visual Basic

Public Function **CreateParameter()** As SAParameter

C#

public SAParameter **CreateParameter();**

Return value

A new parameter, as an SAParameter object.

Remarks

Stored procedures and some other SQL statements can take parameters, indicated in the text of a statement by a question mark (?).

The CreateParameter method provides an SAParameter object. You can set properties on the SAParameter to specify the value, data type, and so on for the parameter.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "SAParameter class" on page 359

EndExecuteNonQuery method

Finishes asynchronous execution of a SQL statement or stored procedure.

Syntax

Visual Basic

```
Public Function EndExecuteNonQuery( _
   ByVal asyncResult As IAsyncResult _
) As Integer
```

C#

```
public int EndExecuteNonQuery(
   IAsyncResult asyncResult
);
```

Parameters

- **asyncResult** The IAsyncResult returned by the call to SACommand.BeginExecuteNonQuery.

Return value

The number of rows affected (the same behavior as SACommand.ExecuteNonQuery).

Remarks

You must call EndExecuteNonQuery once for every call to BeginExecuteNonQuery. The call must be after BeginExecuteNonQuery has returned. ADO.NET is not thread safe; it is your responsibility to ensure that BeginExecuteNonQuery has returned. The IAsyncResult passed to EndExecuteNonQuery must be the same as the one returned from the BeginExecuteNonQuery call that is being completed. It is an error to call EndExecuteNonQuery to end a call to BeginExecuteReader, and vice versa.

If an error occurs while executing the command, the exception is thrown when EndExecuteNonQuery is called.

There are four ways to wait for execution to complete:

(1) Call EndExecuteNonQuery.

Calling EndExecuteNonQuery blocks until the command completes. For example:

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 10 Demo");
conn.Open();
SACommand cmd = new SACommand(
  "UPDATE Departments"
    + " SET DepartmentName = 'Engineering'"
    + " WHERE DepartmentID=100",
  conn );
IAsyncResult res = cmd.BeginExecuteNonQuery();
// perform other work
// this will block until the command completes
int rowCount reader = cmd.EndExecuteNonQuery( res );
```

(2) Poll the IsCompleted property of the IAsyncResult.

You can poll the IsCompleted property of the IAsyncResult. For example:

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 10 Demo");
conn.Open();
SACommand cmd = new SACommand(
  "UPDATE Departments"
    + " SET DepartmentName = 'Engineering'"
    + " WHERE DepartmentID=100",
    conn
  );
IAsyncResult res = cmd.BeginExecuteNonQuery();
while( !res.IsCompleted ) {
// do other work
}
// this will not block because the command is finished
int rowCount = cmd.EndExecuteNonQuery( res );
```

(3) Use the IAsyncResult.AsyncWaitHandle property to get a synchronization object.

You can use the IAsyncResult.AsyncWaitHandle property to get a synchronization object, and wait on that. For example:

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 10 Demo");
conn.Open();
SACommand cmd = new SACommand(
  "UPDATE Departments"
    + " SET DepartmentName = 'Engineering'"
    + " WHERE DepartmentID=100",
    conn
  );
IAsyncResult res = cmd.BeginExecuteNonQuery();
// perform other work
WaitHandle wh = res.AsyncWaitHandle;
wh.WaitOne();
// this will not block because the command is finished
int rowCount = cmd.EndExecuteNonQuery( res );
```

(4) Specify a callback function when calling BeginExecuteNonQuery.

You can specify a callback function when calling BeginExecuteNonQuery. For example:

```
private void callbackFunction( IAsyncResult ar )
{
```

```
    SACommand cmd = (SACommand) ar.AsyncState;
    // this won't block since the command has completed
            int rowCount = cmd.EndExecuteNonQuery();
}
// elsewhere in the code
private void DoStuff()
{
      SAConnection conn = new SAConnection("DSN=SQL Anywhere 10 Demo");
      conn.Open();
            SACommand cmd = new SACommand(
        "UPDATE Departments"
    + " SET DepartmentName = 'Engineering'"
    + " WHERE DepartmentID=100",
    conn
  );
   IAsyncResult res = cmd.BeginExecuteNonQuery( callbackFunction, cmd );
   // perform other work.  The callback function will be
   // called when the command completes
}
```

The callback function executes in a separate thread, so the usual caveats related to updating the user interface in a threaded program apply.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "BeginExecuteNonQuery() method" on page 201

EndExecuteReader method

Finishes asynchronous execution of a SQL statement or stored procedure, returning the requested SADataReader.

Syntax

Visual Basic

```
Public Function EndExecuteReader( _
  ByVal asyncResult As IAsyncResult _
) As SADataReader
```

C#

```
public SADataReader EndExecuteReader(
  IAsyncResult asyncResult
);
```

Parameters

- **asyncResult** The IAsyncResult returned by the call to SACommand.BeginExecuteReader.

Return value

An SADataReader object that can be used to retrieve the requested rows (the same behavior as SACommand.ExecuteReader).

Remarks

You must call EndExecuteReader once for every call to BeginExecuteReader. The call must be after BeginExecuteReader has returned. ADO.NET is not thread safe; it is your responsibility to ensure that BeginExecuteReader has returned. The IAsyncResult passed to EndExecuteReader must be the same as the one returned from the BeginExecuteReader call that is being completed. It is an error to call EndExecuteReader to end a call to BeginExecuteNonQuery, and vice versa.

If an error occurs while executing the command, the exception is thrown when EndExecuteReader is called.

There are four ways to wait for execution to complete:

(1) Call EndExecuteReader.

Calling EndExecuteReader blocks until the command completes. For example:

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 10 Demo");
conn.Open();
SACommand cmd = new SACommand( "SELECT * FROM Departments",
  conn );
IAsyncResult res = cmd.BeginExecuteReader();
// perform other work
// this will block until the command completes
SADataReader reader = cmd.EndExecuteReader( res );
```

(2) Poll the IsCompleted property of the IAsyncResult.

You can poll the IsCompleted property of the IAsyncResult. For example:

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 10 Demo");
conn.Open();
SACommand cmd = new SACommand( "SELECT * FROM Departments",
  conn );
IAsyncResult res = cmd.BeginExecuteReader();
while( !res.IsCompleted ) {
// do other work
}
// this will not block because the command is finished
SADataReader reader = cmd.EndExecuteReader( res );
```

(3) Use the IAsyncResult.AsyncWaitHandle property to get a synchronization object.

You can use the IAsyncResult.AsyncWaitHandle property to get a synchronization object, and wait on that. For example:

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 10 Demo");
conn.Open();
SACommand cmd = new SACommand( "SELECT * FROM Departments",
  conn );
IAsyncResult res = cmd.BeginExecuteReader();
// perform other work
WaitHandle wh = res.AsyncWaitHandle;
wh.WaitOne();
// this will not block because the command is finished
SADataReader reader = cmd.EndExecuteReader( res );
```

(4) Specify a callback function when calling BeginExecuteReader

You can specify a callback function when calling BeginExecuteReader. For example:

```
private void callbackFunction( IAsyncResult ar )
{
   SACommand cmd = (SACommand) ar.AsyncState;
   // this won't block since the command has completed
            SADataReader reader = cmd.EndExecuteReader();
}
// elsewhere in the code
private void DoStuff()
{
      SAConnection conn = new SAConnection("DSN=SQL Anywhere 10 Demo");
      conn.Open();
            SACommand cmd = new SACommand( "SELECT * FROM Departments",
        conn );
   IAsyncResult res = cmd.BeginExecuteReader( callbackFunction, cmd );
   // perform other work.  The callback function will be
   // called when the command completes
}
```

The callback function executes in a separate thread, so the usual caveats related to updating the user interface in a threaded program apply.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "BeginExecuteReader() method" on page 202
- "SADataReader class" on page 289

ExecuteNonQuery method

Executes a statement that does not return a result set, such as an INSERT, UPDATE, DELETE, or data definition statement.

Syntax

Visual Basic

Public Overrides Function **ExecuteNonQuery()** As Integer

C#

public override int **ExecuteNonQuery();**

Return value

The number of rows affected.

Remarks

You can use ExecuteNonQuery to change the data in a database without using a DataSet. Do this by executing UPDATE, INSERT, or DELETE statements.

Although ExecuteNonQuery does not return any rows, output parameters or return values that are mapped to parameters are populated with data.

For UPDATE, INSERT, and DELETE statements, the return value is the number of rows affected by the command. For all other types of statements, and for rollbacks, the return value is -1.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "ExecuteReader() method" on page 211

ExecuteReader methods

Executes a SQL statement that returns a result set.

ExecuteReader() method

Executes a SQL statement that returns a result set.

Syntax

Visual Basic

Public Function **ExecuteReader()** As SADataReader

C#

public SADataReader **ExecuteReader();**

Return value

The result set as an SADataReader object.

Remarks

The statement is the current SACommand object, with CommandText and Parameters as needed. The SADataReader object is a read-only, forward-only result set. For modifiable result sets, use an SADataAdapter.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "ExecuteReader methods" on page 211
- "ExecuteNonQuery method" on page 210
- "SADataReader class" on page 289
- "SADataAdapter class" on page 278
- "CommandText property" on page 196
- "Parameters property" on page 199

ExecuteReader(CommandBehavior) method

Executes a SQL statement that returns a result set.

Syntax

Visual Basic

```
Public Function ExecuteReader( _
  ByVal behavior As CommandBehavior _
) As SADataReader
```

C#

```
public SADataReader ExecuteReader(
  CommandBehavior behavior
);
```

Parameters

- **behavior** One of CloseConnection, Default, KeyInfo, SchemaOnly, SequentialAccess, SingleResult, or SingleRow.

  For more information about this parameter, see the .NET Framework documentation for CommandBehavior Enumeration.

Return value

The result set as an SADataReader object.

Remarks

The statement is the current SACommand object, with CommandText and Parameters as needed. The SADataReader object is a read-only, forward-only result set. For modifiable result sets, use an SADataAdapter.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "ExecuteReader methods" on page 211
- "ExecuteNonQuery method" on page 210
- "SADataReader class" on page 289
- "SADataAdapter class" on page 278
- "CommandText property" on page 196
- "Parameters property" on page 199

ExecuteScalar method

Executes a statement that returns a single value. If this method is called on a query that returns multiple rows and columns, only the first column of the first row is returned.

Syntax

Visual Basic

Public Overrides Function **ExecuteScalar()** As Object

C#

public override object **ExecuteScalar();**

Return value

The first column of the first row in the result set, or a null reference if the result set is empty.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192

Prepare method

Prepares or compiles the SACommand on the data source.

Syntax

Visual Basic

Public Overrides Sub **Prepare()**

C#

public override void **Prepare();**

Remarks

If you call one of the ExecuteNonQuery, ExecuteReader, or ExecuteScalar methods after calling Prepare, any parameter value that is larger than the value specified by the Size property is automatically truncated to the original specified size of the parameter, and no truncation errors are returned.

The truncation only happens for the following data types:

- CHAR
- VARCHAR
- LONG VARCHAR
- TEXT
- NCHAR
- NVARCHAR
- LONG NVARCHAR
- NTEXT
- BINARY
- LONG BINARY
- VARBINARY
- IMAGE

If the size property is not specified, and so is using the default value, the data is not truncated.

See also

- "SACommand class" on page 192
- "SACommand members" on page 192
- "ExecuteNonQuery method" on page 210
- "ExecuteReader() method" on page 211
- "ExecuteScalar method" on page 212

ResetCommandTimeout method

Resets the CommandTimeout property to its default value of 30 seconds.

Syntax

Visual Basic

```
Public Sub ResetCommandTimeout()
```

C#

```
public void ResetCommandTimeout();
```

See also

- "SACommand class" on page 192
- "SACommand members" on page 192

SACommandBuilder class

A way to generate single-table SQL statements that reconcile changes made to a DataSet with the data in the associated database. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SACommandBuilder
  Inherits DbCommandBuilder
```

C#

```
public sealed class SACommandBuilder : DbCommandBuilder
```

See also

- "SACommandBuilder members" on page 215

SACommandBuilder members

Public constructors

Member name	Description
SACommandBuilder constructors	Initializes a new instance of the "SACommandBuilder class" on page 214.

Public properties

Member name	Description
CatalogLocation (inherited from DbCommandBuilder)	Sets or gets the CatalogLocation for an instance of the DbCommandBuilder.
CatalogSeparator (inherited from DbCommandBuilder)	Sets or gets a string used as the catalog separator for an instance of the DbCommandBuilder.
ConflictOption (inherited from DbCommandBuilder)	Specifies which ConflictOption is to be used by the DbCommandBuilder.
DataAdapter property	Specifies the SADataAdapter for which to generate statements.
QuotePrefix (inherited from DbCommandBuilder)	Gets or sets the beginning character or characters to use when specifying database objects (for example, tables or columns) whose names contain characters such as spaces or reserved tokens.
QuoteSuffix (inherited from DbCommandBuilder)	Gets or sets the beginning character or characters to use when specifying database objects (for example, tables or columns) whose names contain characters such as spaces or reserved tokens.
SchemaSeparator (inherited from DbCommandBuilder)	Gets or sets the character to be used for the separator between the schema identifier and any other identifiers.
SetAllValues (inherited from DbCommandBuilder)	Specifies whether all column values in an update statement are included or only changed ones.

Public methods

Member name	Description
DeriveParameters method	Populates the Parameters collection of the specified SACommand object. This is used for the stored procedure specified in the SACommand.
GetDeleteCommand methods	Returns the generated SACommand object that performs DELETE operations on the database when SADataAdapter.Update is called.

Member name	Description
GetInsertCommand methods	Returns the generated SACommand object that performs INSERT operations on the database when an Update is called.
GetUpdateCommand methods	Returns the generated SACommand object that performs UPDATE operations on the database when an Update is called.
QuoteIdentifier method	Returns the correct quoted form of an unquoted identifier, including properly escaping any embedded quotes in the identifier.
RefreshSchema (inherited from DbCommandBuilder)	Clears the commands associated with this DbCommandBuilder.
UnquoteIdentifier method	Returns the correct unquoted form of a quoted identifier, including properly un-escaping any embedded quotes in the identifier.

See also

- "SACommandBuilder class" on page 214

SACommandBuilder constructors

Initializes a new instance of the "SACommandBuilder class" on page 214.

SACommandBuilder() constructor

Initializes an SACommandBuilder object.

Syntax

Visual Basic

Public Sub **New()**

C#

public **SACommandBuilder();**

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215
- "SACommandBuilder constructors" on page 216

SACommandBuilder(SADataAdapter) constructor

Initializes an SACommandBuilder object.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal adapter As SADataAdapter _
)
```

C#

```
public  SACommandBuilder(
  SADataAdapter adapter
);
```

Parameters

- **adapter** An SADataAdapter object for which to generate reconciliation statements.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215
- "SACommandBuilder constructors" on page 216

DataAdapter property

Specifies the SADataAdapter for which to generate statements.

Syntax

Visual Basic

```
Public Property DataAdapter As SADataAdapter
```

C#

```
public SADataAdapter DataAdapter  { get; set; }
```

Property value

An SADataAdapter object.

Remarks

When you create a new instance of SACommandBuilder, any existing SACommandBuilder that is associated with this SADataAdapter is released.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215

DeriveParameters method

Populates the Parameters collection of the specified SACommand object. This is used for the stored procedure specified in the SACommand.

Syntax

Visual Basic

```
Public Shared Sub DeriveParameters( _
  ByVal command As SACommand _
)
```

C#

```
public static void DeriveParameters(
  SACommand command
);
```

Parameters

- **command** An SACommand object for which to derive parameters.

Remarks

DeriveParameters overwrites any existing parameter information for the SACommand.

DeriveParameters requires an extra call to the database server. If the parameter information is known in advance, it is more efficient to populate the Parameters collection by setting the information explicitly.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215

GetDeleteCommand methods

Returns the generated SACommand object that performs DELETE operations on the database when SADataAdapter.Update is called.

GetDeleteCommand(Boolean) method

Returns the generated SACommand object that performs DELETE operations on the database when SADataAdapter.Update is called.

Syntax

Visual Basic

```
Public Function GetDeleteCommand( _
  ByVal useColumnsForParameterNames As Boolean _
) As SACommand
```


C#

```
public SACommand GetDeleteCommand(
  bool useColumnsForParameterNames
);
```

Parameters

- **useColumnsForParameterNames** If true, generate parameter names matching column names if possible. If false, generate @p1, @p2, and so on.

Return value

The automatically generated SACommand object required to perform deletions.

Remarks

The GetDeleteCommand method returns the SACommand object to be executed, so it may be useful for informational or troubleshooting purposes.

You can also use GetDeleteCommand as the basis of a modified command. For example, you might call GetDeleteCommand and modify the CommandTimeout value, and then explicitly set that value on the SADataAdapter.

SQL statements are first generated when the application calls Update or GetDeleteCommand. After the SQL statement is first generated, the application must explicitly call RefreshSchema if it changes the statement in any way. Otherwise, the GetDeleteCommand will still be using information from the previous statement.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215
- "GetDeleteCommand methods" on page 218
- DbCommandBuilder.RefreshSchema

GetDeleteCommand() method

Returns the generated SACommand object that performs DELETE operations on the database when SADataAdapter.Update is called.

Syntax

Visual Basic

```
Public Function GetDeleteCommand() As SACommand
```

C#

```
public SACommand GetDeleteCommand();
```

Return value

The automatically generated SACommand object required to perform deletions.

Remarks

The GetDeleteCommand method returns the SACommand object to be executed, so it may be useful for informational or troubleshooting purposes.

You can also use GetDeleteCommand as the basis of a modified command. For example, you might call GetDeleteCommand and modify the CommandTimeout value, and then explicitly set that value on the SADataAdapter.

SQL statements are first generated when the application calls Update or GetDeleteCommand. After the SQL statement is first generated, the application must explicitly call RefreshSchema if it changes the statement in any way. Otherwise, the GetDeleteCommand will still be using information from the previous statement.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215
- "GetDeleteCommand methods" on page 218
- DbCommandBuilder.RefreshSchema

GetInsertCommand methods

Returns the generated SACommand object that performs INSERT operations on the database when an Update is called.

GetInsertCommand(Boolean) method

Returns the generated SACommand object that performs INSERT operations on the database when an Update is called.

Syntax

Visual Basic

```
Public Function GetInsertCommand( _
  ByVal useColumnsForParameterNames As Boolean _
) As SACommand
```

C#

```
public SACommand GetInsertCommand(
  bool useColumnsForParameterNames
);
```

Parameters

- **useColumnsForParameterNames** If true, generate parameter names matching column names if possible. If false, generate @p1, @p2, and so on.

Return value

The automatically generated SACommand object required to perform insertions.

Remarks

The GetInsertCommand method returns the SACommand object to be executed, so it may be useful for informational or troubleshooting purposes.

You can also use GetInsertCommand as the basis of a modified command. For example, you might call GetInsertCommand and modify the CommandTimeout value, and then explicitly set that value on the SADataAdapter.

SQL statements are first generated either when the application calls Update or GetInsertCommand. After the SQL statement is first generated, the application must explicitly call RefreshSchema if it changes the statement in any way. Otherwise, the GetInsertCommand will be still be using information from the previous statement, which might not be correct.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215
- "GetInsertCommand methods" on page 220
- "GetDeleteCommand() method" on page 219

GetInsertCommand() method

Returns the generated SACommand object that performs INSERT operations on the database when an Update is called.

Syntax

Visual Basic

Public Function **GetInsertCommand()** As SACommand

C#

public SACommand **GetInsertCommand();**

Return value

The automatically generated SACommand object required to perform insertions.

Remarks

The GetInsertCommand method returns the SACommand object to be executed, so it may be useful for informational or troubleshooting purposes.

You can also use GetInsertCommand as the basis of a modified command. For example, you might call GetInsertCommand and modify the CommandTimeout value, and then explicitly set that value on the SADataAdapter.

SQL statements are first generated either when the application calls Update or GetInsertCommand. After the SQL statement is first generated, the application must explicitly call RefreshSchema if it changes the statement in any way. Otherwise, the GetInsertCommand will be still be using information from the previous statement, which might not be correct.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215
- "GetInsertCommand methods" on page 220
- "GetDeleteCommand() method" on page 219

GetUpdateCommand methods

Returns the generated SACommand object that performs UPDATE operations on the database when an Update is called.

GetUpdateCommand(Boolean) method

Returns the generated SACommand object that performs UPDATE operations on the database when an Update is called.

Syntax

Visual Basic

```
Public Function GetUpdateCommand( _
   ByVal useColumnsForParameterNames As Boolean _
) As SACommand
```

C#

```
public SACommand GetUpdateCommand(
   bool useColumnsForParameterNames
);
```

Parameters

- **useColumnsForParameterNames** If true, generate parameter names matching column names if possible. If false, generate @p1, @p2, and so on.

Return value

The automatically generated SACommand object required to perform updates.

Remarks

The GetUpdateCommand method returns the SACommand object to be executed, so it may be useful for informational or troubleshooting purposes.

You can also use GetUpdateCommand as the basis of a modified command. For example, you might call GetUpdateCommand and modify the CommandTimeout value, and then explicitly set that value on the SADataAdapter.

SQL statements are first generated when the application calls Update or GetUpdateCommand. After the SQL statement is first generated, the application must explicitly call RefreshSchema if it changes the

statement in any way. Otherwise, the GetUpdateCommand will be still be using information from the previous statement, which might not be correct.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215
- "GetUpdateCommand methods" on page 222
- DbCommandBuilder.RefreshSchema

GetUpdateCommand() method

Returns the generated SACommand object that performs UPDATE operations on the database when an Update is called.

Syntax

Visual Basic

Public Function **GetUpdateCommand()** As SACommand

C#

public SACommand **GetUpdateCommand();**

Return value

The automatically generated SACommand object required to perform updates.

Remarks

The GetUpdateCommand method returns the SACommand object to be executed, so it may be useful for informational or troubleshooting purposes.

You can also use GetUpdateCommand as the basis of a modified command. For example, you might call GetUpdateCommand and modify the CommandTimeout value, and then explicitly set that value on the SADataAdapter.

SQL statements are first generated when the application calls Update or GetUpdateCommand. After the SQL statement is first generated, the application must explicitly call RefreshSchema if it changes the statement in any way. Otherwise, the GetUpdateCommand will be still be using information from the previous statement, which might not be correct.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215
- "GetUpdateCommand methods" on page 222
- DbCommandBuilder.RefreshSchema

QuoteIdentifier method

Returns the correct quoted form of an unquoted identifier, including properly escaping any embedded quotes in the identifier.

Syntax

Visual Basic

```
Public Overrides Function QuoteIdentifier( _
  ByVal unquotedIdentifier As String _
) As String
```

C#

```
public override string  QuoteIdentifier(
  string  unquotedIdentifier
);
```

Parameters

- **unquotedIdentifier** The string representing the unquoted identifier that will have be quoted.

Return value

Returns a string representing the quoted form of an unquoted identifier with embedded quotes properly escaped.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215

UnquoteIdentifier method

Returns the correct unquoted form of a quoted identifier, including properly un-escaping any embedded quotes in the identifier.

Syntax

Visual Basic

```
Public Overrides Function UnquoteIdentifier( _
  ByVal quotedIdentifier As String _
) As String
```

C#

```
public override string  UnquoteIdentifier(
  string  quotedIdentifier
);
```


Parameters

- **quotedIdentifier** The string representing the quoted identifier that will have its embedded quotes removed.

Return value

Returns a string representing the unquoted form of a quoted identifier with embedded quotes properly un-escaped.

See also

- "SACommandBuilder class" on page 214
- "SACommandBuilder members" on page 215

SACommLinksOptionsBuilder class

Provides a simple way to create and manage the CommLinks options portion of connection strings used by the SAConnection class. This class cannot be inherited.

Syntax

Visual Basic

Public NotInheritable Class **SACommLinksOptionsBuilder**

C#

public sealed class **SACommLinksOptionsBuilder**

Remarks

The SACommLinksOptionsBuilder class is not available in the .NET Compact Framework 2.0.

For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

See also

- "SACommLinksOptionsBuilder members" on page 225

SACommLinksOptionsBuilder members

Public constructors

Member name	Description
SACommLinksOptionsBuilder constructors	Initializes a new instance of the "SACommLinksOptionsBuilder class" on page 225.

Public properties

Member name	Description
All property	Gets or sets the ALL CommLinks option.
ConnectionString property	Gets or sets the connection string being built.
SharedMemory property	Gets or sets the SharedMemory protocol.
TcpOptionsBuilder property	Gets or sets a TcpOptionsBuilder object used to create a TCP options string.
TcpOptionsString property	Gets or sets a string of TCP options.

Public methods

Member name	Description
GetUseLongNameAsKeyword method	Gets a boolean values that indicates whether long connection parameter names are used in the connection string.
SetUseLongNameAsKeyword method	Sets a boolean value that indicates whether long connection parameter names are used in the connection string. Long connection parameter names are used by default.
ToString method	Converts the SACommLinksOptionsBuilder object to a string representation.

See also

- "SACommLinksOptionsBuilder class" on page 225

SACommLinksOptionsBuilder constructors

Initializes a new instance of the "SACommLinksOptionsBuilder class" on page 225.

SACommLinksOptionsBuilder() constructor

Initializes an SACommLinksOptionsBuilder object.

Syntax

Visual Basic

Public Sub **New()**

C#

public **SACommLinksOptionsBuilder();**

Remarks

The SACommLinksOptionsBuilder class is not available in the .NET Compact Framework 2.0.

Example

The following statement initializes an SACommLinksOptionsBuilder object.

```
SACommLinksOptionsBuilder commLinks =
 new SACommLinksOptionsBuilder( );
```

See also

- "SACommLinksOptionsBuilder class" on page 225
- "SACommLinksOptionsBuilder members" on page 225
- "SACommLinksOptionsBuilder constructors" on page 226

SACommLinksOptionsBuilder(String) constructor

Initializes an SACommLinksOptionsBuilder object.

Syntax

Visual Basic

Public Sub **New(** _
  ByVal *options* As String _
)

C#

public **SACommLinksOptionsBuilder(**
  string *options*
);

Parameters

- **options** A SQL Anywhere CommLinks connection parameter string.

  For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Remarks

The SACommLinksOptionsBuilder class is not available in the .NET Compact Framework 2.0.

Example

The following statement initializes an SACommLinksOptionsBuilder object.

```
SACommLinksOptionsBuilder commLinks =
 new SACommLinksOptionsBuilder("TCPIP(DoBroadcast=ALL;Timeout=20)" );
```

See also

- "SACommLinksOptionsBuilder class" on page 225
- "SACommLinksOptionsBuilder members" on page 225
- "SACommLinksOptionsBuilder constructors" on page 226

All property

Gets or sets the ALL CommLinks option.

Syntax

Visual Basic

Public Property **All** As Boolean

C#

public bool **All** { get; set; }

Remarks

Attempt to connect using the shared memory protocol first, followed by all remaining and available communication protocols. Use this setting if you are unsure of which communication protocol(s) to use.

The SACommLinksOptionsBuilder class is not available in the .NET Compact Framework 2.0.

See also

- "SACommLinksOptionsBuilder class" on page 225
- "SACommLinksOptionsBuilder members" on page 225

ConnectionString property

Gets or sets the connection string being built.

Syntax

Visual Basic

Public Property **ConnectionString** As String

C#

public string **ConnectionString** { get; set; }

Remarks

The SACommLinksOptionsBuilder class is not available in the .NET Compact Framework 2.0.

See also

- "SACommLinksOptionsBuilder class" on page 225
- "SACommLinksOptionsBuilder members" on page 225

SharedMemory property

Gets or sets the SharedMemory protocol.

Syntax

Visual Basic

Public Property **SharedMemory** As Boolean

C#

public bool **SharedMemory** { get; set; }

Remarks

The SACommLinksOptionsBuilder class is not available in the .NET Compact Framework 2.0.

See also

- "SACommLinksOptionsBuilder class" on page 225
- "SACommLinksOptionsBuilder members" on page 225

TcpOptionsBuilder property

Gets or sets a TcpOptionsBuilder object used to create a TCP options string.

Syntax

Visual Basic

Public Property **TcpOptionsBuilder** As SATcpOptionsBuilder

C#

public SATcpOptionsBuilder **TcpOptionsBuilder** { get; set; }

See also

- "SACommLinksOptionsBuilder class" on page 225
- "SACommLinksOptionsBuilder members" on page 225

TcpOptionsString property

Gets or sets a string of TCP options.

Syntax

Visual Basic

Public Property **TcpOptionsString** As String

C#

```
public string  TcpOptionsString  { get; set; }
```

See also

- "SACommLinksOptionsBuilder class" on page 225
- "SACommLinksOptionsBuilder members" on page 225

GetUseLongNameAsKeyword method

Gets a boolean values that indicates whether long connection parameter names are used in the connection string.

Syntax

Visual Basic

```
Public Function GetUseLongNameAsKeyword() As Boolean
```

C#

```
public bool GetUseLongNameAsKeyword();
```

Return value

True if long connection parameter names are used to build connection strings; otherwise, false.

Remarks

SQL Anywhere connection parameters have both long and short forms of their names. For example, to specify the name of an ODBC data source in your connection string, you can use either of the following values: DataSourceName or DSN. By default, long connection parameter names are used to build connection strings.

See also

- "SACommLinksOptionsBuilder class" on page 225
- "SACommLinksOptionsBuilder members" on page 225
- "SetUseLongNameAsKeyword method" on page 230

SetUseLongNameAsKeyword method

Sets a boolean value that indicates whether long connection parameter names are used in the connection string. Long connection parameter names are used by default.

Syntax

Visual Basic

```
Public Sub SetUseLongNameAsKeyword( _
   ByVal useLongNameAsKeyword As Boolean _
)
```


C#

```
public void SetUseLongNameAsKeyword(
  bool useLongNameAsKeyword
);
```

Parameters

- **useLongNameAsKeyword** A boolean value that indicates whether the long connection parameter name is used in the connection string.

See also

- "SACommLinksOptionsBuilder class" on page 225
- "SACommLinksOptionsBuilder members" on page 225
- "GetUseLongNameAsKeyword method" on page 230

ToString method

Converts the SACommLinksOptionsBuilder object to a string representation.

Syntax

Visual Basic

```
Public Overrides Function ToString() As String
```

C#

```
public override string  ToString();
```

Return value

The options string being built.

See also

- "SACommLinksOptionsBuilder class" on page 225
- "SACommLinksOptionsBuilder members" on page 225

SAConnection class

Represents a connection to a SQL Anywhere database. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SAConnection
  Inherits DbConnection
```

C#

```
public sealed class SAConnection : DbConnection
```

Remarks

For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

See also

- "SAConnection members" on page 232

SAConnection members

Public constructors

Member name	Description
SAConnection constructors	Initializes a new instance of the "SAConnection class" on page 231.

Public properties

Member name	Description
ConnectionString property	Provides the database connection string.
ConnectionTimeout property	Gets the number of seconds before a connection attempt times out with an error.
DataSource property	Gets the name of the database server.
Database property	Gets the name of the current database.
InitString property	A command that is executed immediately after the connection is established.
ServerVersion property	Gets a string that contains the version of the instance of SQL Anywhere to which the client is connected.
State property	Indicates the state of the SAConnection object.

Public methods

Member name	Description
BeginTransaction methods	Returns a transaction object. Commands associated with a transaction object are executed as a single transaction. The transaction is terminated with a call to the Commit or Rollback methods.
ChangeDatabase method	Changes the current database for an open SAConnection.

Member name	Description
ChangePassword method	Changes the password for the user indicated in the connection string to the supplied new password.
ClearAllPools method	Empties all connection pools.
ClearPool method	Empties the connection pool associated with the specified connection.
Close method	Closes a database connection.
CreateCommand method	Initializes an SACommand object.
EnlistDistributedTransaction method	Enlists in the specified transaction as a distributed transaction.
EnlistTransaction method	Enlists in the specified transaction as a distributed transaction.
GetSchema methods	Returns the list of supported schema collections.
Open method	Opens a database connection with the property settings specified by the SAConnection.ConnectionString.

Public events

Member name	Description
InfoMessage event	Occurs when the SQL Anywhere database server returns a warning or informational message.
StateChange event	Occurs when the state of the SAConnection object changes.

See also

- "SAConnection class" on page 231

SAConnection constructors

Initializes a new instance of the "SAConnection class" on page 231.

SAConnection() constructor

Initializes an SAConnection object. The connection must be opened before you can perform any operations against the database.

Syntax

Visual Basic

Public Sub **New()**

C#

public **SAConnection();**

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "SAConnection constructors" on page 233

SAConnection(String) constructor

Initializes an SAConnection object. The connection must then be opened before you can perform any operations against the database.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal connectionString As String _
)
```

C#

```
public SAConnection(
  string connectionString
);
```

Parameters

- **connectionString** A SQL Anywhere connection string. A connection string is a semicolon-separated list of keyword=value pairs.

  For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Example

The following statement initializes an SAConnection object for a connection to a database named policies running on a SQL Anywhere database server named hr. The connection uses the user ID admin and the password money.

```
SAConnection conn = new SAConnection(
"UID=admin;PWD=money;ENG=hr;DBN=policies" );
conn.Open();
```

See also

- "SAConnection class" on page 231

- "SAConnection members" on page 232
- "SAConnection constructors" on page 233
- "SAConnection class" on page 231

ConnectionString property

Provides the database connection string.

Syntax

Visual Basic

```
Public Overrides Property ConnectionString As String
```

C#

```
public override string  ConnectionString  { get; set; }
```

Remarks

The ConnectionString is designed to match the SQL Anywhere connection string format as closely as possible with the following exception: when the Persist Security Info value is set to false (the default), the connection string that is returned is the same as the user-set ConnectionString minus security information. The SQL Anywhere SQL Anywhere .NET Data Provider does not persist the password in a returned connection string unless you set Persist Security Info to true.

You can use the ConnectionString property to connect to a variety of data sources.

You can set the ConnectionString property only when the connection is closed. Many of the connection string values have corresponding read-only properties. When the connection string is set, all of these properties are updated, unless an error is detected. If an error is detected, none of the properties are updated. SAConnection properties return only those settings contained in the ConnectionString.

If you reset the ConnectionString on a closed connection, all connection string values and related properties are reset, including the password.

When the property is set, a preliminary validation of the connection string is performed. When an application calls the Open method, the connection string is fully validated. A runtime exception is generated if the connection string contains invalid or unsupported properties.

Values can be delimited by single or double quotes. Either single or double quotes may be used within a connection string by using the other delimiter, for example, name="value's" or name= 'value''s', but not name='value's' or name= ""value"". Blank characters are ignored unless they are placed within a value or within quotes. keyword=value pairs must be separated by a semicolon. If a semicolon is part of a value, it must also be delimited by quotes. Escape sequences are not supported, and the value type is irrelevant. Names are not case sensitive. If a property name occurs more than once in the connection string, the value associated with the last occurrence is used.

You should use caution when constructing a connection string based on user input, such as when retrieving a user ID and password from a dialog, and appending it to the connection string. The application should not allow a user to embed extra connection string parameters in these values.

The default value of connection pooling is true (pooling=true).

Example

The following statements set a connection string for an ODBC data source named SQL Anywhere 10 Demo and open the connection.

```
SAConnection conn = new SAConnection();
conn.ConnectionString = "DSN=SQL Anywhere 10 Demo";
conn.Open();
```

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "SAConnection class" on page 231
- "Open method" on page 247

ConnectionTimeout property

Gets the number of seconds before a connection attempt times out with an error.

Syntax

Visual Basic

Public Overrides Readonly Property **ConnectionTimeout** As Integer

C#

public override int **ConnectionTimeout** { get;}

Property value

15 seconds

Example

The following statement displays the value of the ConnectionTimeout.

```
MessageBox.Show( conn.ConnectionTimeout.ToString( ) );
```

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

DataSource property

Gets the name of the database server.

Syntax

Visual Basic

Public Overrides Readonly Property **DataSource** As String

C#

public override string **DataSource** { get;}

Remarks

If the connection is opened, the SAConnection object returns the ServerName server property. Otherwise, the SAConnection object looks in the connection string in the following order: EngineName, ServerName, ENG.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "SAConnection class" on page 231

Database property

Gets the name of the current database.

Syntax

Visual Basic

Public Overrides Readonly Property **Database** As String

C#

public override string **Database** { get;}

Remarks

If the connection is opened, SAConnection returns the name of the current database. Otherwise, SAConnection looks in the connection string in the following order: DatabaseName, DBN, DataSourceName, DataSource, DSN, DatabaseFile, DBF.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

InitString property

A command that is executed immediately after the connection is established.

Syntax

Visual Basic

Public Property **InitString** As String

C#

public string **InitString** { get; set; }

Remarks

The InitString will be executed immediately after the connection is opened.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

ServerVersion property

Gets a string that contains the version of the instance of SQL Anywhere to which the client is connected.

Syntax

Visual Basic

Public Overrides Readonly Property **ServerVersion** As String

C#

public override string **ServerVersion** { get;}

Property value

The version of the instance of SQL Anywhere.

Remarks

The version is ##.##.####, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version. The appended string is of the form major.minor.build, where major and minor are two digits, and build is four digits.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

State property

Indicates the state of the SAConnection object.

Syntax

Visual Basic

Public Overrides Readonly Property **State** As ConnectionState

C#

public override ConnectionState **State** { get;}

Property value

A ConnectionState enumeration.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

BeginTransaction methods

Returns a transaction object. Commands associated with a transaction object are executed as a single transaction. The transaction is terminated with a call to the Commit or Rollback methods.

BeginTransaction() method

Returns a transaction object. Commands associated with a transaction object are executed as a single transaction. The transaction is terminated with a call to the Commit or Rollback methods.

Syntax

Visual Basic

Public Function **BeginTransaction()** As SATransaction

C#

public SATransaction **BeginTransaction();**

Return value

An SATransaction object representing the new transaction.

Remarks

To associate a command with a transaction object, use the SACommand.Transaction property.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "BeginTransaction methods" on page 239
- "SATransaction class" on page 416
- "Transaction property" on page 199

BeginTransaction(IsolationLevel) method

Returns a transaction object. Commands associated with a transaction object are executed as a single transaction. The transaction is terminated with a call to the Commit or Rollback methods.

Syntax

Visual Basic

```
Public Function BeginTransaction( _
  ByVal isolationLevel As IsolationLevel _
) As SATransaction
```

C#

```
public SATransaction BeginTransaction(
  IsolationLevel isolationLevel
);
```

Parameters

- **isolationLevel** A member of the SAIsolationLevel enumeration. The default value is ReadCommitted.

Return value

An SATransaction object representing the new transaction.

Remarks

To associate a command with a transaction object, use the SACommand.Transaction property.

Example

```
SATransaction tx = conn.BeginTransaction(
    SAIsolationLevel.ReadUncommitted );
```

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "BeginTransaction methods" on page 239
- "SATransaction class" on page 416
- "Transaction property" on page 199
- "SAIsolationLevel enumeration" on page 348

BeginTransaction(SAIsolationLevel) method

Returns a transaction object. Commands associated with a transaction object are executed as a single transaction. The transaction is terminated with a call to the Commit or Rollback methods.

Syntax

Visual Basic

```
Public Function BeginTransaction( _
```



```
  ByVal isolationLevel As SAIsolationLevel _
) As SATransaction
```

C#

```
public SATransaction BeginTransaction(
  SAIsolationLevel isolationLevel
);
```

Parameters

- **isolationLevel** A member of the SAIsolationLevel enumeration. The default value is ReadCommitted.

Return value

An SATransaction object representing the new transaction.

For more information, see "Transaction processing" on page 134.

For more information, see "Typical types of inconsistency" [*SQL Anywhere Server - SQL Usage*].

Remarks

To associate a command with a transaction object, use the SACommand.Transaction property.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "BeginTransaction methods" on page 239
- "SATransaction class" on page 416
- "Transaction property" on page 199
- "SAIsolationLevel enumeration" on page 348
- "Commit method" on page 419
- "Rollback() method" on page 420
- "Rollback(String) method" on page 420

ChangeDatabase method

Changes the current database for an open SAConnection.

Syntax

Visual Basic

```
Public Overrides Sub ChangeDatabase( _
  ByVal database As String _
)
```

C#

```
public override void ChangeDatabase(
  string  database
);
```

Parameters

- **database** The name of the database to use instead of the current database.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

ChangePassword method

Changes the password for the user indicated in the connection string to the supplied new password.

Syntax

Visual Basic

```
Public Shared Sub ChangePassword( _
   ByVal connectionString As String, _
   ByVal newPassword As String _
)
```

C#

```
public static void ChangePassword(
   string  connectionString,
   string  newPassword
);
```

Parameters

- **connectionString** The connection string that contains enough information to connect to the database server that you want. The connection string may contain the user ID and the current password.
- **newPassword** The new password to set. This password must comply with any password security policy set on the server, including minimum length, requirements for specific characters, and so on.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

ClearAllPools method

Empties all connection pools.

Syntax

Visual Basic

```
Public Shared Sub ClearAllPools()
```


C#

```
public static void ClearAllPools();
```

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

ClearPool method

Empties the connection pool associated with the specified connection.

Syntax

Visual Basic

```
Public Shared Sub ClearPool( _
  ByVal connection As SAConnection _
)
```

C#

```
public static void ClearPool(
  SAConnection connection
);
```

Parameters

- **connection** The SAConnection object to be cleared from the pool.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "SAConnection class" on page 231

Close method

Closes a database connection.

Syntax

Visual Basic

```
Public Overrides Sub Close()
```

C#

```
public override void Close();
```

Remarks

The Close method rolls back any pending transactions. It then releases the connection to the connection pool, or closes the connection if connection pooling is disabled. If Close is called while handling a StateChange event, no additional StateChange events are fired. An application can call Close multiple times.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

CreateCommand method

Initializes an SACommand object.

Syntax

Visual Basic

```
Public Function CreateCommand() As SACommand
```

C#

```
public SACommand CreateCommand();
```

Return value

An SACommand object.

Remarks

The command object is associated with the SAConnection object.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "SACommand class" on page 192
- "SAConnection class" on page 231

EnlistDistributedTransaction method

Enlists in the specified transaction as a distributed transaction.

Syntax

Visual Basic

```
Public Sub EnlistDistributedTransaction( _
  ByVal transaction As ITransaction _
)
```

C#


```
public void EnlistDistributedTransaction(
  ITransaction transaction
);
```

Parameters

- **transaction** A reference to an existing System.EnterpriseServices.ITransaction in which to enlist.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

EnlistTransaction method

Enlists in the specified transaction as a distributed transaction.

Syntax

Visual Basic

```
Public Overrides Sub EnlistTransaction( _
  ByVal transaction As Transaction _
)
```

C#

```
public override void EnlistTransaction(
  Transaction transaction
);
```

Parameters

- **transaction** A reference to an existing System.Transactions.Transaction in which to enlist.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

GetSchema methods

Returns the list of supported schema collections.

GetSchema() method

Returns the list of supported schema collections.

Syntax

Visual Basic

```
Public Overrides Function GetSchema() As DataTable
```

C#

```
public override DataTable GetSchema();
```

Remarks

See GetSchema(string,string[]) for a description of the available metadata.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "GetSchema methods" on page 245
- "GetSchema(String, String[]) method" on page 246

GetSchema(String) method

Returns the list of supported schema collections.

Syntax

Visual Basic

```
Public Overrides Function GetSchema( _
  ByVal collection As String _
) As DataTable
```

C#

```
public override DataTable GetSchema(
  string  collection
);
```

Remarks

See GetSchema(string,string[]) for a description of the available metadata.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "GetSchema methods" on page 245
- "GetSchema(String, String[]) method" on page 246

GetSchema(String, String[]) method

Returns the list of supported schema collections.

Syntax

Visual Basic

```
Public Overrides Function GetSchema( _
  ByVal collection As String, _
  ByVal restrictions As String() _
) As DataTable
```

C#

```
public override DataTable GetSchema(
  string  collection,
  string [] restrictions
);
```

Remarks

See GetSchema(string,string[]) for a description of the available metadata.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "GetSchema methods" on page 245
- "GetSchema(String, String[]) method" on page 246

Open method

Opens a database connection with the property settings specified by the SAConnection.ConnectionString.

Syntax

Visual Basic

```
Public Overrides Sub Open()
```

C#

```
public override void Open();
```

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232
- "ConnectionString property" on page 235

InfoMessage event

Occurs when the SQL Anywhere database server returns a warning or informational message.

Syntax

Visual Basic

Public Event **InfoMessage** As SAInfoMessageEventHandler

C#

public event SAInfoMessageEventHandler **InfoMessage** ;

Remarks

The event handler receives an argument of type SaInfoMessageEventArgs containing data related to this event. The following SAaInfoMessageEventArgs properties provide information specific to this event: NativeError, Errors, Message, MessageType, and Source.

For more information, see the .NET Framework documentation for OleDbConnection.InfoMessage Event.

Event data

- **MessageType** Returns the type of the message. This can be one of: Action, Info, Status, or Warning.
- **Errors** Returns the collection of messages sent from the data source.
- **Message** Returns the full text of the error sent from the data source.
- **Source** Returns the name of the SQL Anywhere .NET Data Provider.
- **NativeError** Returns the SQL code returned by the database.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

StateChange event

Occurs when the state of the SAConnection object changes.

Syntax

Visual Basic

Public Overrides Event **StateChange** As StateChangeEventHandler

C#

public event override StateChangeEventHandler **StateChange** ;

Remarks

The event handler receives an argument of type StateChangeEventArgs with data related to this event. The following StateChangeEventArgs properties provide information specific to this event: CurrentState and OriginalState.

For more information, see the .NET Framework documentation for OleDbConnection.StateChange Event.

Event data

- **CurrentState** Gets the new state of the connection. The connection object will be in the new state already when the event is fired.
- **OriginalState** Gets the original state of the connection.

See also

- "SAConnection class" on page 231
- "SAConnection members" on page 232

SAConnectionStringBuilder class

Provides a simple way to create and manage the contents of connection strings used by the SAConnection class. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SAConnectionStringBuilder
  Inherits SAConnectionStringBuilderBase
```

C#

```
public sealed class SAConnectionStringBuilder : SAConnectionstring BuilderBase
```

Remarks

The SA ConnectionStringBuilder class inherits SAConnectionStringBuilderBase, which inherits DbConnectionStringBuilder.

Restrictions: The SAConnectionStringBuilder class is not available in the .NET Compact Framework 2.0.

Inherits:"SAConnectionStringBuilderBase class" on page 271

For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

See also

- "SAConnectionStringBuilder members" on page 249

SAConnectionStringBuilder members

Public constructors

Member name	Description
SAConnectionStringBuilder constructors	Initializes a new instance of the "SAConnectionStringBuilder class" on page 249.

Public properties

Member name	Description
AppInfo property	Gets or sets the AppInfo connection property.
AutoStart property	Gets or sets the AutoStart connection property.
AutoStop property	Gets or sets the AutoStop connection property.
BrowsableConnectionString (inherited from DbConnectionStringBuilder)	Gets or sets a value that indicates whether the DbConnectionStringBuilder.ConnectionString is visible in Visual Studio designers.
Charset property	Gets or sets the Charset connection property.
CommBufferSize property	Gets or sets the CommBufferSize connection property.
CommLinks property	Gets or sets the CommLinks property.
Compress property	Gets or sets the Compress connection property.
CompressionThreshold property	Gets or sets the CompressionThreshold connection property.
ConnectionLifetime property	Gets or sets the ConnectionLifetime connection property.
ConnectionName property	Gets or sets the ConnectionName connection property.
ConnectionReset property	Gets or sets the ConnectionReset connection property.
ConnectionString (inherited from DbConnectionStringBuilder)	Gets or sets the connection string associated with the DbConnectionStringBuilder.
ConnectionTimeout property	Gets or sets the ConnectionTimeout connection property.
Count (inherited from DbConnectionStringBuilder)	Gets the current number of keys that are contained within the DbConnectionStringBuilder.ConnectionString.
DataSourceName property	Gets or sets the DataSourceName connection property.
DatabaseFile property	Gets or sets the DatabaseFile connection property.
DatabaseKey property	Gets or sets the DatabaseKey connection property.
DatabaseName property	Gets or sets the DatabaseName connection property.
DatabaseSwitches property	Gets or sets the DatabaseSwitches connection property.
DisableMultiRowFetch property	Gets or sets the DisableMultiRowFetch connection property.

Member name	Description
Elevate property	Gets or sets the Elevate connection property.
EncryptedPassword property	Gets or sets the EncryptedPassword connection property.
Encryption property	Gets or sets the Encryption connection property.
Enlist property	Gets or sets the Enlist connection property.
FileDataSourceName property	Gets or sets the FileDataSourceName connection property.
ForceStart property	Gets or sets the ForceStart connection property.
IdleTimeout property	Gets or sets the IdleTimeout connection property.
Integrated property	Gets or sets the Integrated connection property.
IsFixedSize (inherited from DbConnectionStringBuilder)	Gets a value that indicates whether the DbConnectionStringBuilder has a fixed size.
IsReadOnly (inherited from DbConnectionStringBuilder)	Gets a value that indicates whether the DbConnectionStringBuilder is read-only.
Item property (inherited from SAConnectionStringBuilder-Base)	Gets or sets the value of the connection keyword.
Kerberos property	Gets or sets the Kerberos connection property.
Keys property (inherited from SAConnectionStringBuilder-Base)	Gets an System.Collections.ICollection that contains the keys in the SAConnectionStringBuilder.
Language property	Gets or sets the Language connection property.
LazyClose property	Gets or sets the LazyClose connection property.
LivenessTimeout property	Gets or sets the LivenessTimeout connection property.
LogFile property	Gets or sets the LogFile connection property.
MaxPoolSize property	Gets or sets the MaxPoolSize connection property.
MinPoolSize property	Gets or sets the MinPoolSize connection property.
NewPassword property	Gets or sets the NewPassword connection property.
Password property	Gets or sets the Password connection property.

Member name	Description
PersistSecurityInfo property	Gets or sets the PersistSecurityInfo connection property.
Pooling property	Gets or sets the Pooling connection property.
PrefetchBuffer property	Gets or sets the PrefetchBuffer connection property.
PrefetchRows property	Gets or sets the PrefetchRows connection property. The default value is 200.
RetryConnectionTimeout property	Gets or sets the RetryConnectionTimeout property.
ServerName property	Gets or sets the ServerName connection property.
StartLine property	Gets or sets the StartLine connection property.
Unconditional property	Gets or sets the Unconditional connection property.
UserID property	Gets or sets the UserID connection property.
Values (inherited from DbConnectionStringBuilder)	Gets an ICollection that contains the values in the DbConnectionStringBuilder.

Public methods

Member name	Description
Add (inherited from DbConnectionStringBuilder)	Adds an entry with the specified key and value into the DbConnectionStringBuilder.
Clear (inherited from DbConnectionStringBuilder)	Clears the contents of the DbConnectionStringBuilder instance.
ContainsKey method (inherited from SAConnectionStringBuilderBase)	Determines whether the SAConnectionStringBuilder object contains a specific keyword.
EquivalentTo (inherited from DbConnectionStringBuilder)	Compares the connection information in this DbConnectionStringBuilder object with the connection information in the supplied object.
GetKeyword method (inherited from SAConnectionStringBuilderBase)	Gets the keyword for specified SAConnectionStringBuilder property.
GetUseLongNameAsKeyword method (inherited from SAConnectionStringBuilderBase)	Gets a boolean values that indicates whether long connection parameter names are used in the connection string.

Member name	Description
Remove method (inherited from SAConnectionStringBuilder-Base)	Removes the entry with the specified key from the SAConnection-StringBuilder instance.
SetUseLongNameAsKeyword method (inherited from SACon-nectionStringBuilderBase)	Sets a boolean value that indicates whether long connection parameter names are used in the connection string. Long connection parameter names are used by default.
ShouldSerialize method (inheri-ted from SAConnectionString-BuilderBase)	Indicates whether the specified key exists in this SAConnection-StringBuilder instance.
ToString (inherited from DbCon-nectionStringBuilder)	Returns the connection string associated with this DbConnection-StringBuilder.
TryGetValue method (inherited from SAConnectionStringBuil-derBase)	Retrieves a value corresponding to the supplied key from this SA-ConnectionStringBuilder.

See also

- "SAConnectionStringBuilder class" on page 249

SAConnectionStringBuilder constructors

Initializes a new instance of the "SAConnectionStringBuilder class" on page 249.

SAConnectionStringBuilder() constructor

Initializes a new instance of the SAConnectionStringBuilder class.

Syntax

Visual Basic

Public Sub **New()**

C#

public **SAConnectionStringBuilder();**

Remarks

Restrictions: The SAConnectionStringBuilder class is not available in the .NET Compact Framework 2.0.

See also

- "SAConnectionStringBuilder class" on page 249

- "SAConnectionStringBuilder members" on page 249
- "SAConnectionStringBuilder constructors" on page 253

SAConnectionStringBuilder(String) constructor

Initializes a new instance of the SAConnectionStringBuilder class.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal connectionString As String _
)
```

C#

```
public  SAConnectionStringBuilder(
  string  connectionString
);
```

Parameters

- **connectionString** The basis for the object's internal connection information. Parsed into keyword=value pairs.

  For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Remarks

Restrictions: The SAConnectionStringBuilder class is not available in the .NET Compact Framework 2.0.

Example

The following statement initializes an SAConnection object for a connection to a database named policies running on a SQL Anywhere database server named hr. The connection uses the user ID admin and the password money.

```
SAConnectionStringBuilder conn = new
SAConnectionStringBuilder("UID=admin;PWD=money;ENG=hr;DBN=policies" );
```

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249
- "SAConnectionStringBuilder constructors" on page 253

AppInfo property

Gets or sets the AppInfo connection property.

Syntax

Visual Basic

Public Property **AppInfo** As String

C#

public string **AppInfo** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

AutoStart property

Gets or sets the AutoStart connection property.

Syntax

Visual Basic

Public Property **AutoStart** As String

C#

public string **AutoStart** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

AutoStop property

Gets or sets the AutoStop connection property.

Syntax

Visual Basic

Public Property **AutoStop** As String

C#

public string **AutoStop** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Charset property

Gets or sets the Charset connection property.

Syntax

Visual Basic

Public Property **Charset** As String

C#

public string **Charset** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

CommBufferSize property

Gets or sets the CommBufferSize connection property.

Syntax

Visual Basic

Public Property **CommBufferSize** As Integer

C#

public int **CommBufferSize** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

CommLinks property

Gets or sets the CommLinks property.

Syntax

Visual Basic

Public Property **CommLinks** As String

C#

public string **CommLinks** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Compress property

Gets or sets the Compress connection property.

Syntax

Visual Basic

Public Property **Compress** As String

C#

public string **Compress** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

CompressionThreshold property

Gets or sets the CompressionThreshold connection property.

Syntax

Visual Basic

Public Property **CompressionThreshold** As Integer

C#

public int **CompressionThreshold** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

ConnectionLifetime property

Gets or sets the ConnectionLifetime connection property.

Syntax

Visual Basic

Public Property **ConnectionLifetime** As Integer

C#

public int **ConnectionLifetime** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

ConnectionName property

Gets or sets the ConnectionName connection property.

Syntax

Visual Basic

Public Property **ConnectionName** As String

C#

public string **ConnectionName** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

ConnectionReset property

Gets or sets the ConnectionReset connection property.

Syntax

Visual Basic

Public Property **ConnectionReset** As Boolean

C#

public bool **ConnectionReset** { get; set; }

Property value

A DataTable that contains schema information.

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

ConnectionTimeout property

Gets or sets the ConnectionTimeout connection property.

Syntax

Visual Basic

Public Property **ConnectionTimeout** As Integer

C#

public int **ConnectionTimeout** { get; set; }

Example

The following statement displays the value of the ConnectionTimeout property.

```
MessageBox.Show( connString.ConnectionTimeout.ToString() );
```

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

DataSourceName property

Gets or sets the DataSourceName connection property.

Syntax

Visual Basic

Public Property **DataSourceName** As String

C#

public string **DataSourceName** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

DatabaseFile property

Gets or sets the DatabaseFile connection property.

Syntax

Visual Basic

Public Property **DatabaseFile** As String

C#

```
public string  DatabaseFile  { get; set; }
```

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

DatabaseKey property

Gets or sets the DatabaseKey connection property.

Syntax

Visual Basic

```
Public Property DatabaseKey As String
```

C#

```
public string  DatabaseKey  { get; set; }
```

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

DatabaseName property

Gets or sets the DatabaseName connection property.

Syntax

Visual Basic

```
Public Property DatabaseName As String
```

C#

```
public string  DatabaseName  { get; set; }
```

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

DatabaseSwitches property

Gets or sets the DatabaseSwitches connection property.

Syntax

Visual Basic

Public Property **DatabaseSwitches** As String

C#

public string **DatabaseSwitches** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

DisableMultiRowFetch property

Gets or sets the DisableMultiRowFetch connection property.

Syntax

Visual Basic

Public Property **DisableMultiRowFetch** As String

C#

public string **DisableMultiRowFetch** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Elevate property

Gets or sets the Elevate connection property.

Syntax

Visual Basic

Public Property **Elevate** As String

C#

public string **Elevate** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

EncryptedPassword property

Gets or sets the EncryptedPassword connection property.

Syntax

Visual Basic

Public Property **EncryptedPassword** As String

C#

public string **EncryptedPassword** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Encryption property

Gets or sets the Encryption connection property.

Syntax

Visual Basic

Public Property **Encryption** As String

C#

public string **Encryption** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Enlist property

Gets or sets the Enlist connection property.

Syntax

Visual Basic

Public Property **Enlist** As Boolean

C#

public bool **Enlist** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

FileDataSourceName property

Gets or sets the FileDataSourceName connection property.

Syntax

Visual Basic

Public Property **FileDataSourceName** As String

C#

public string **FileDataSourceName** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

ForceStart property

Gets or sets the ForceStart connection property.

Syntax

Visual Basic

Public Property **ForceStart** As String

C#

public string **ForceStart** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

IdleTimeout property

Gets or sets the IdleTimeout connection property.

Syntax

Visual Basic

Public Property **IdleTimeout** As Integer

C#

public int **IdleTimeout** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Integrated property

Gets or sets the Integrated connection property.

Syntax

Visual Basic

Public Property **Integrated** As String

C#

public string **Integrated** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Kerberos property

Gets or sets the Kerberos connection property.

Syntax

Visual Basic

Public Property **Kerberos** As String

C#

public string **Kerberos** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Language property

Gets or sets the Language connection property.

Syntax

Visual Basic

Public Property **Language** As String

C#

public string **Language** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

LazyClose property

Gets or sets the LazyClose connection property.

Syntax

Visual Basic

Public Property **LazyClose** As String

C#

public string **LazyClose** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

LivenessTimeout property

Gets or sets the LivenessTimeout connection property.

Syntax

Visual Basic

Public Property **LivenessTimeout** As Integer

C#

public int **LivenessTimeout** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

LogFile property

Gets or sets the LogFile connection property.

Syntax

Visual Basic

Public Property **LogFile** As String

C#

public string **LogFile** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

MaxPoolSize property

Gets or sets the MaxPoolSize connection property.

Syntax

Visual Basic

Public Property **MaxPoolSize** As Integer

C#

public int **MaxPoolSize** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

MinPoolSize property

Gets or sets the MinPoolSize connection property.

Syntax

Visual Basic

Public Property **MinPoolSize** As Integer

C#

public int **MinPoolSize** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

NewPassword property

Gets or sets the NewPassword connection property.

Syntax

Visual Basic

Public Property **NewPassword** As String

C#

public string **NewPassword** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Password property

Gets or sets the Password connection property.

Syntax

Visual Basic

Public Property **Password** As String

C#

public string **Password** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

PersistSecurityInfo property

Gets or sets the PersistSecurityInfo connection property.

Syntax

Visual Basic

Public Property **PersistSecurityInfo** As Boolean

C#

public bool **PersistSecurityInfo** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Pooling property

Gets or sets the Pooling connection property.

Syntax

Visual Basic

Public Property **Pooling** As Boolean

C#

public bool **Pooling** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

PrefetchBuffer property

Gets or sets the PrefetchBuffer connection property.

Syntax

Visual Basic

Public Property **PrefetchBuffer** As Integer

C#

public int **PrefetchBuffer** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

PrefetchRows property

Gets or sets the PrefetchRows connection property. The default value is 200.

Syntax

Visual Basic

Public Property **PrefetchRows** As Integer

C#

public int **PrefetchRows** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

RetryConnectionTimeout property

Gets or sets the RetryConnectionTimeout property.

Syntax

Visual Basic

Public Property **RetryConnectionTimeout** As Integer

C#

public int **RetryConnectionTimeout** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

ServerName property

Gets or sets the ServerName connection property.

Syntax

Visual Basic

Public Property **ServerName** As String

C#

public string **ServerName** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

StartLine property

Gets or sets the StartLine connection property.

Syntax

Visual Basic

Public Property **StartLine** As String

C#

public string **StartLine** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

Unconditional property

Gets or sets the Unconditional connection property.

Syntax

Visual Basic

Public Property **Unconditional** As String

C#

public string **Unconditional** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

UserID property

Gets or sets the UserID connection property.

Syntax

Visual Basic

Public Property **UserID** As String

C#

public string **UserID** { get; set; }

See also

- "SAConnectionStringBuilder class" on page 249
- "SAConnectionStringBuilder members" on page 249

SAConnectionStringBuilderBase class

Base class of the SAConnectionStringBuilder class. This class is abstract and so cannot be instantiated.

Syntax

Visual Basic

```
MustInherit Public Class SAConnectionStringBuilderBase
  Inherits DbConnectionStringBuilder
```

C#

```
public abstract class SAConnectionStringBuilderBase : DbConnectionstring Builder
```

See also

- "SAConnectionStringBuilderBase members" on page 271

SAConnectionStringBuilderBase members

Public properties

Member name	Description
BrowsableConnectionString (inherited from DbConnectionStringBuilder)	Gets or sets a value that indicates whether the DbConnectionStringBuilder.ConnectionString is visible in Visual Studio designers.
ConnectionString (inherited from DbConnectionStringBuilder)	Gets or sets the connection string associated with the DbConnectionStringBuilder.
Count (inherited from DbConnectionStringBuilder)	Gets the current number of keys that are contained within the DbConnectionStringBuilder.ConnectionString.
IsFixedSize (inherited from DbConnectionStringBuilder)	Gets a value that indicates whether the DbConnectionStringBuilder has a fixed size.
IsReadOnly (inherited from DbConnectionStringBuilder)	Gets a value that indicates whether the DbConnectionStringBuilder is read-only.
Item property	Gets or sets the value of the connection keyword.

Member name	Description
Keys property	Gets an System.Collections.ICollection that contains the keys in the SAConnectionStringBuilder.
Values (inherited from DbConnectionStringBuilder)	Gets an ICollection that contains the values in the DbConnectionStringBuilder.

Public methods

Member name	Description
Add (inherited from DbConnectionStringBuilder)	Adds an entry with the specified key and value into the DbConnectionStringBuilder.
Clear (inherited from DbConnectionStringBuilder)	Clears the contents of the DbConnectionStringBuilder instance.
ContainsKey method	Determines whether the SAConnectionStringBuilder object contains a specific keyword.
EquivalentTo (inherited from DbConnectionStringBuilder)	Compares the connection information in this DbConnectionStringBuilder object with the connection information in the supplied object.
GetKeyword method	Gets the keyword for specified SAConnectionStringBuilder property.
GetUseLongNameAsKeyword method	Gets a boolean values that indicates whether long connection parameter names are used in the connection string.
Remove method	Removes the entry with the specified key from the SAConnectionStringBuilder instance.
SetUseLongNameAsKeyword method	Sets a boolean value that indicates whether long connection parameter names are used in the connection string. Long connection parameter names are used by default.
ShouldSerialize method	Indicates whether the specified key exists in this SAConnectionStringBuilder instance.
ToString (inherited from DbConnectionStringBuilder)	Returns the connection string associated with this DbConnectionStringBuilder.
TryGetValue method	Retrieves a value corresponding to the supplied key from this SAConnectionStringBuilder.

See also

- "SAConnectionStringBuilderBase class" on page 271

Item property

Gets or sets the value of the connection keyword.

Syntax

Visual Basic

```
Public Overrides Default Property Item ( _
  ByVal keyword As String _
) As Object
```

C#

```
public override object this [
  string  keyword
] { get; set; }
```

Parameters

- **keyword** The name of the connection keyword.

Property value

An object representing the value of the specified connection keyword.

Remarks

If the keyword or type is invalid, an exception is raised. keyword is case insensitive.

When setting the value, passing NULL clears the value.

See also

- "SAConnectionStringBuilderBase class" on page 271
- "SAConnectionStringBuilderBase members" on page 271

Keys property

Gets an System.Collections.ICollection that contains the keys in the SAConnectionStringBuilder.

Syntax

Visual Basic

```
Public Overrides Readonly Property Keys As ICollection
```

C#

```
public override ICollection Keys  { get;}
```

Property value

An System.Collections.ICollection that contains the keys in the SAConnectionStringBuilder.

See also

- "SAConnectionStringBuilderBase class" on page 271
- "SAConnectionStringBuilderBase members" on page 271

ContainsKey method

Determines whether the SAConnectionStringBuilder object contains a specific keyword.

Syntax

Visual Basic

```
Public Overrides Function ContainsKey( _
  ByVal keyword As String _
) As Boolean
```

C#

```
public override bool ContainsKey(
  string  keyword
);
```

Parameters

- **keyword** The keyword to locate in the SAConnectionStringBuilder.

Return value

True if the value associated with keyword has been set; otherwise, false.

Example

The following statement determines whether the SAConnectionStringBuilder object contains the UserID keyword.

```
connectString.ContainsKey("UserID")
```

See also

- "SAConnectionStringBuilderBase class" on page 271
- "SAConnectionStringBuilderBase members" on page 271

GetKeyword method

Gets the keyword for specified SAConnectionStringBuilder property.

Syntax

Visual Basic

```
Public Function GetKeyword( _
  ByVal propName As String _
) As String
```


C#

```
public string  GetKeyword(
   string  propName
);
```

Parameters

- **propName** The name of the SAConnectionStringBuilder property.

Return value

The keyword for specified SAConnectionStringBuilder property.

See also

- "SAConnectionStringBuilderBase class" on page 271
- "SAConnectionStringBuilderBase members" on page 271

GetUseLongNameAsKeyword method

Gets a boolean values that indicates whether long connection parameter names are used in the connection string.

Syntax

Visual Basic

```
Public Function GetUseLongNameAsKeyword() As Boolean
```

C#

```
public bool GetUseLongNameAsKeyword();
```

Return value

True if long connection parameter names are used to build connection strings; otherwise, false.

Remarks

SQL Anywhere connection parameters have both long and short forms of their names. For example, to specify the name of an ODBC data source in your connection string, you can use either of the following values: DataSourceName or DSN. By default, long connection parameter names are used to build connection strings.

See also

- "SAConnectionStringBuilderBase class" on page 271
- "SAConnectionStringBuilderBase members" on page 271
- "SetUseLongNameAsKeyword method" on page 276

Remove method

Removes the entry with the specified key from the SAConnectionStringBuilder instance.

Syntax

Visual Basic

```
Public Overrides Function Remove( _
  ByVal keyword As String _
) As Boolean
```

C#

```
public override bool Remove(
  string  keyword
);
```

Parameters

- **keyword** The key of the key/value pair to be removed from the connection string in this SAConnectionStringBuilder.

Return value

True if the key existed within the connection string and was removed; false if the key did not exist.

See also

- "SAConnectionStringBuilderBase class" on page 271
- "SAConnectionStringBuilderBase members" on page 271

SetUseLongNameAsKeyword method

Sets a boolean value that indicates whether long connection parameter names are used in the connection string. Long connection parameter names are used by default.

Syntax

Visual Basic

```
Public Sub SetUseLongNameAsKeyword( _
  ByVal useLongNameAsKeyword As Boolean _
)
```

C#

```
public void SetUseLongNameAsKeyword(
  bool useLongNameAsKeyword
);
```

Parameters

- **useLongNameAsKeyword** A boolean value that indicates whether the long connection parameter name is used in the connection string.

See also

- "SAConnectionStringBuilderBase class" on page 271
- "SAConnectionStringBuilderBase members" on page 271
- "GetUseLongNameAsKeyword method" on page 275

ShouldSerialize method

Indicates whether the specified key exists in this SAConnectionStringBuilder instance.

Syntax

Visual Basic

```
Public Overrides Function ShouldSerialize( _
  ByVal keyword As String _
) As Boolean
```

C#

```
public override bool ShouldSerialize(
  string  keyword
);
```

Parameters

- **keyword** The key to locate in the SAConnectionStringBuilder.

Return value

True if the SAConnectionStringBuilder contains an entry with the specified key; otherwise false.

See also

- "SAConnectionStringBuilderBase class" on page 271
- "SAConnectionStringBuilderBase members" on page 271

TryGetValue method

Retrieves a value corresponding to the supplied key from this SAConnectionStringBuilder.

Syntax

Visual Basic

```
Public Overrides Function TryGetValue( _
  ByVal keyword As String, _
  ByVal value As Object _
) As Boolean
```

C#

```
public override bool TryGetValue(
  string  keyword,
```

```
  object value
);
```

Parameters

- **keyword** The key of the item to retrieve.
- **value** The value corresponding to keyword.

Return value

true if keyword was found within the connection string; otherwise false.

See also

- "SAConnectionStringBuilderBase class" on page 271
- "SAConnectionStringBuilderBase members" on page 271

SADataAdapter class

Represents a set of commands and a database connection used to fill a DataSet and to update a database. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SADataAdapter
  Inherits DbDataAdapter
```

C#

```
public sealed class SADataAdapter : DbDataAdapter
```

Remarks

The DataSet provides a way to work with data offline. The SADataAdapter provides methods to associate a DataSet with a set of SQL statements.

Implements:IDbDataAdapter, IDataAdapter, ICloneable

For more information, see "Using the SADataAdapter object to access and manipulate data" on page 121 and "Accessing and manipulating data" on page 115.

See also

- "SADataAdapter members" on page 279

SADataAdapter members

Public constructors

Member name	Description
SADataAdapter constructors	Initializes a new instance of the "SADataAdapter class" on page 278.

Public properties

Member name	Description
AcceptChangesDuringFill (inherited from DataAdapter)	Gets or sets a value indicating whether DataRow.AcceptChanges is called on a DataRow after it is added to the DataTable during any of the Fill operations.
AcceptChangesDuringUpdate (inherited from DataAdapter)	Gets or sets whether DataRow.AcceptChanges is called during a DataAdapter.Update.
ContinueUpdateOnError (inherited from DataAdapter)	Gets or sets a value that specifies whether to generate an exception when an error is encountered during a row update.
DeleteCommand property	Specifies an SACommand object that is executed against the database when the Update method is called to delete rows in the database that correspond to deleted rows in the DataSet.
FillLoadOption (inherited from DataAdapter)	Gets or sets the LoadOption that determines how the adapter fills the DataTable from the DbDataReader.
InsertCommand property	Specifies an SACommand that is executed against the database when the Update method is called that adds rows to the database to correspond to rows that were inserted in the DataSet.
MissingMappingAction (inherited from DataAdapter)	Determines the action to take when incoming data does not have a matching table or column.
MissingSchemaAction (inherited from DataAdapter)	Determines the action to take when existing DataSet schema does not match incoming data.
ReturnProviderSpecificTypes (inherited from DataAdapter)	Gets or sets whether the Fill method should return provider-specific values or common CLS-compliant values.
SelectCommand property	Specifies an SACommand that is used during Fill or FillSchema to obtain a result set from the database for copying into a DataSet.
TableMappings property	Specifies a collection that provides the master mapping between a source table and a DataTable.

Member name	Description
UpdateBatchSize property	Gets or sets the number of rows that are processed in each round-trip to the server.
UpdateCommand property	Specifies an SACommand that is executed against the database when the Update method is called to update rows in the database that correspond to updated rows in the DataSet.

Public methods

Member name	Description
Fill (inherited from DbDataAdapter)	Adds or refreshes rows in the DataSet.
FillSchema (inherited from DbDataAdapter)	Adds a DataTable named "Table" to the specified DataSet and configures the schema to match that in the data source based on the specified SchemaType.
GetFillParameters method	Returns the parameters set by you when executing a SELECT statement.
ResetFillLoadOption (inherited from DataAdapter)	Resets DataAdapter.FillLoadOption to its default state and causes DataAdapter.Fill to honor DataAdapter.AcceptChangesDuringFill.
ShouldSerializeAcceptChangesDuringFill (inherited from DataAdapter)	Determines whether the DataAdapter.AcceptChangesDuringFill should be persisted.
ShouldSerializeFillLoadOption (inherited from DataAdapter)	Determines whether the DataAdapter.FillLoadOption should be persisted.
Update (inherited from DbDataAdapter)	Calls the respective INSERT, UPDATE, or DELETE statements for each inserted, updated, or deleted row in the specified array of DataRow objects.

Public events

Member name	Description
FillError (inherited from DataAdapter)	Returned when an error occurs during a fill operation.
RowUpdated event	Occurs during an update after a command is executed against the data source. When an attempt to update is made, the event fires.

Member name	Description
RowUpdating event	Occurs during an update before a command is executed against the data source. When an attempt to update is made, the event fires.

See also

- "SADataAdapter class" on page 278

SADataAdapter constructors

Initializes a new instance of the "SADataAdapter class" on page 278.

SADataAdapter() constructor

Initializes an SADataAdapter object.

Syntax

Visual Basic

```
Public Sub New()
```

C#

```
public  SADataAdapter();
```

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279
- "SADataAdapter constructors" on page 281
- "SADataAdapter(SACommand) constructor" on page 281
- "SADataAdapter(String, SAConnection) constructor" on page 282
- "SADataAdapter(String, String) constructor" on page 283

SADataAdapter(SACommand) constructor

Initializes an SADataAdapter object with the specified SELECT statement.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal selectCommand As SACommand _
)
```

C#

```
public  SADataAdapter(
  SACommand selectCommand
);
```

Parameters

- **selectCommand** An SACommand object that is used during DbDataAdapter.Fill to select records from the data source for placement in the DataSet.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279
- "SADataAdapter constructors" on page 281
- "SADataAdapter() constructor" on page 281
- "SADataAdapter(String, SAConnection) constructor" on page 282
- "SADataAdapter(String, String) constructor" on page 283

SADataAdapter(String, SAConnection) constructor

Initializes an SADataAdapter object with the specified SELECT statement and connection.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal selectCommandText As String, _
  ByVal selectConnection As SAConnection _
)
```

C#

```
public  SADataAdapter(
  string  selectCommandText,
  SAConnection selectConnection
);
```

Parameters

- **selectCommandText** A SELECT statement to be used to set the SADataAdapter.SelectCommand property of the SADataAdapter object.
- **selectConnection** An SAConnection object that defines a connection to a database.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279
- "SADataAdapter constructors" on page 281
- "SADataAdapter() constructor" on page 281
- "SADataAdapter(SACommand) constructor" on page 281
- "SADataAdapter(String, String) constructor" on page 283

- "SelectCommand property" on page 285
- "SAConnection class" on page 231

SADataAdapter(String, String) constructor

Initializes an SADataAdapter object with the specified SELECT statement and connection string.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal selectCommandText As String, _
  ByVal selectConnectionString As String _
)
```

C#

```
public  SADataAdapter(
  string  selectCommandText,
  string  selectConnectionString
);
```

Parameters

- **selectCommandText** A SELECT statement to be used to set the SADataAdapter.SelectCommand property of the SADataAdapter object.
- **selectConnectionString** A connection string for a SQL Anywhere database.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279
- "SADataAdapter constructors" on page 281
- "SADataAdapter() constructor" on page 281
- "SADataAdapter(SACommand) constructor" on page 281
- "SADataAdapter(String, SAConnection) constructor" on page 282
- "SelectCommand property" on page 285

DeleteCommand property

Specifies an SACommand object that is executed against the database when the Update method is called to delete rows in the database that correspond to deleted rows in the DataSet.

Syntax

Visual Basic

```
Public Property DeleteCommand As SACommand
```

C#

public SACommand **DeleteCommand** { get; set; }

Remarks

If this property is not set and primary key information is present in the DataSet during Update, DeleteCommand can be generated automatically by setting SelectCommand and using the SACommandBuilder. In that case, the SACommandBuilder generates any additional commands that you do not set. This generation logic requires key column information to be present in the SelectCommand.

When DeleteCommand is assigned to an existing SACommand object, the SACommand object is not cloned. The DeleteCommand maintains a reference to the existing SACommand.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279
- "SelectCommand property" on page 285

InsertCommand property

Specifies an SACommand that is executed against the database when the Update method is called that adds rows to the database to correspond to rows that were inserted in the DataSet.

Syntax

Visual Basic

Public Property **InsertCommand** As SACommand

C#

public SACommand **InsertCommand** { get; set; }

Remarks

The SACommandBuilder does not require key columns to generate InsertCommand.

When InsertCommand is assigned to an existing SACommand object, the SACommand is not cloned. The InsertCommand maintains a reference to the existing SACommand.

If this command returns rows, the rows may be added to the DataSet depending on how you set the UpdatedRowSource property of the SACommand object.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279

SelectCommand property

Specifies an SACommand that is used during Fill or FillSchema to obtain a result set from the database for copying into a DataSet.

Syntax

Visual Basic

Public Property **SelectCommand** As SACommand

C#

public SACommand **SelectCommand** { get; set; }

Remarks

When SelectCommand is assigned to a previously-created SACommand, the SACommand is not cloned. The SelectCommand maintains a reference to the previously-created SACommand object.

If the SelectCommand does not return any rows, no tables are added to the DataSet, and no exception is raised.

The SELECT statement can also be specified in the SADataAdapter constructor.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279

TableMappings property

Specifies a collection that provides the master mapping between a source table and a DataTable.

Syntax

Visual Basic

Public Readonly Property **TableMappings** As DataTableMappingCollection

C#

public DataTableMappingCollection **TableMappings** { get;}

Remarks

The default value is an empty collection.

When reconciling changes, the SADataAdapter uses the DataTableMappingCollection collection to associate the column names used by the data source with the column names used by the DataSet.

Restrictions: The TableMappings property is not available in the .NET Compact Framework 2.0.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279

UpdateBatchSize property

Gets or sets the number of rows that are processed in each round-trip to the server.

Syntax

Visual Basic

Public Overrides Property **UpdateBatchSize** As Integer

C#

public override int **UpdateBatchSize** { get; set; }

Remarks

Setting the value of UpdateBatchSize to 1 causes SADataAdapter.Fill to send its rows as in version 1.0 of the .NET Data Provider: each row in the input set is updated one at a time, in order. The default value is 1.

Setting the value to something greater than 1 causes SADataAdapter.Fill to execute all the insert statements in batches. The deletions and updates are executed sequentially as before, but insertions are executed afterward in batches of size equal to the value of UpdateBatchSize.

Setting the value to 0 causes Fill to send the insert statements in a single batch.

Setting it less than 0 is an error.

If UpdateBatchSize is set to something other than one, and the InsertCommand property is set to something that is not an INSERT statement, then an exception is thrown when calling Fill.

This behavior is different from SqlDataAdapter. It batches all types of commands.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279

UpdateCommand property

Specifies an SACommand that is executed against the database when the Update method is called to update rows in the database that correspond to updated rows in the DataSet.

Syntax

Visual Basic

Public Property **UpdateCommand** As SACommand

C#

public SACommand **UpdateCommand** { get; set; }

Remarks

During Update, if this property is not set and primary key information is present in the SelectCommand, the UpdateCommand can be generated automatically if you set the SelectCommand property and use the SACommandBuilder. Then, any additional commands that you do not set are generated by the SACommandBuilder. This generation logic requires key column information to be present in the SelectCommand.

When UpdateCommand is assigned to a previously-created SACommand, the SACommand is not cloned. The UpdateCommand maintains a reference to the previously-created SACommand object.

If execution of this command returns rows, these rows can be merged with the DataSet depending on how you set the UpdatedRowSource property of the SACommand object.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279

GetFillParameters method

Returns the parameters set by you when executing a SELECT statement.

Syntax

Visual Basic

Public Function **GetFillParameters()** As SAParameter

C#

public SAParameter **GetFillParameters();**

Return value

An array of IDataParameter objects that contains the parameters set by the user.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279

RowUpdated event

Occurs during an update after a command is executed against the data source. When an attempt to update is made, the event fires.

Syntax

Visual Basic

Public Event **RowUpdated** As SARowUpdatedEventHandler

C#

public event SARowUpdatedEventHandler **RowUpdated** ;

Remarks

The event handler receives an argument of type SARowUpdatedEventArgs containing data related to this event.

For more information, see the .NET Framework documentation for OleDbDataAdapter.RowUpdated Event.

Event data

- **Command** Gets the SACommand that is executed when DataAdapter.Update is called.
- **RecordsAffected** Returns the number of rows changed, inserted, or deleted by execution of the SQL statement.
- **Command** Gets the IDbCommand executed when DbDataAdapter.Update is called.
- **Errors** Gets any errors generated by the .NET Framework data provider when the RowUpdatedEventArgs.Command was executed.
- **Row** Gets the DataRow sent through an DbDataAdapter.Update.
- **RowCount** Gets the number of rows processed in a batch of updated records.
- **StatementType** Gets the type of SQL statement executed.
- **Status** Gets the UpdateStatus of the RowUpdatedEventArgs.Command.
- **TableMapping** Gets the DataTableMapping sent through an DbDataAdapter.Update.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279

RowUpdating event

Occurs during an update before a command is executed against the data source. When an attempt to update is made, the event fires.

Syntax

Visual Basic

Public Event **RowUpdating** As SARowUpdatingEventHandler

C#

public event SARowUpdatingEventHandler **RowUpdating** ;

Remarks

The event handler receives an argument of type SARowUpdatingEventArgs containing data related to this event.

For more information, see the .NET Framework documentation for OleDbDataAdapter.RowUpdating Event.

Event data

- **Command** Specifies the SACommand to execute when performing the Update.
- **Command** Gets the IDbCommand to execute during the DbDataAdapter.Update operation.
- **Errors** Gets any errors generated by the .NET Framework data provider when the RowUpdatedEventArgs.Command executes.
- **Row** Gets the DataRow that will be sent to the server as part of an insert, update, or delete operation.
- **StatementType** Gets the type of SQL statement to execute.
- **Status** Gets or sets the UpdateStatus of the RowUpdatedEventArgs.Command.
- **TableMapping** Gets the DataTableMapping to send through the DbDataAdapter.Update.

See also

- "SADataAdapter class" on page 278
- "SADataAdapter members" on page 279

SADataReader class

A read-only, forward-only result set from a query or stored procedure. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SADataReader
  Inherits DbDataReader
  Implements IListSource
```

C#

```
public sealed class SADataReader : DbDataReader,
  IListSource
```

Remarks

There is no constructor for SADataReader. To get an SADataReader object, execute an SACommand:

```
SACommand cmd = new SACommand(
    "SELECT EmployeeID FROM Employees", conn );
SADataReader reader = cmd.ExecuteReader();
```

You can only move forward through an SADataReader. If you need a more flexible object to manipulate results, use an SADataAdapter.

The SADataReader retrieves rows as needed, whereas the SADataAdapter must retrieve all rows of a result set before you can carry out any action on the object. For large result sets, this difference gives the SADataReader a much faster response time.

Implements:IDataReader, IDisposable, IDataRecord, IListSource

For more information, see "Accessing and manipulating data" on page 115.

See also

- "SADataReader members" on page 290
- "ExecuteReader() method" on page 211

SADataReader members

Public properties

Member name	Description
Depth property	Gets a value indicating the depth of nesting for the current row. The outermost table has a depth of zero.
FieldCount property	Gets the number of columns in the result set.
HasRows property	Gets a value that indicates whether the SADataReader contains one or more rows.
IsClosed property	Gets a values that indicates whether the SADataReader is closed.
Item properties	Returns the value of a column in its native format. In C#, this property is the indexer for the SADataReader class.
RecordsAffected property	The number of rows changed, inserted, or deleted by execution of the SQL statement.
VisibleFieldCount (inherited from DbDataReader)	Gets the number of fields in the DbDataReader that are not hidden.

Public methods

Member name	Description
Close method	Closes the SADataReader.
Dispose (inherited from DbDataReader)	Releases all resources used by the current instance of the DbDataReader.
GetBoolean method	Returns the value of the specified column as a Boolean.

Member name	Description
GetByte method	Returns the value of the specified column as a Byte.
GetBytes method	Reads a stream of bytes from the specified column offset into the buffer as an array, starting at the given buffer offset.
GetChar method	Returns the value of the specified column as a character.
GetChars method	Reads a stream of characters from the specified column offset into the buffer as an array starting at the given buffer offset.
GetData method	This method is not supported. When called, it throws an InvalidOperationException.
GetDataTypeName method	Returns the name of the source data type.
GetDateTime method	Returns the value of the specified column as a DateTime object.
GetDecimal method	Returns the value of the specified column as a Decimal object.
GetDouble method	Returns the value of the specified column as a double-precision floating point number.
GetEnumerator method	Returns a IEnumerator that iterates through the SADataReader object.
GetFieldType method	Returns the Type that is the data type of the object.
GetFloat method	Returns the value of the specified column as a single-precision floating point number.
GetGuid method	Returns the value of the specified column as a global unique identifier (GUID).
GetInt16 method	Returns the value of the specified column as a 16-bit signed integer.
GetInt32 method	Returns the value of the specified column as a 32-bit signed integer.
GetInt64 method	Returns the value of the specified column as a 64-bit signed integer.
GetName method	Returns the name of the specified column.
GetOrdinal method	Returns the column ordinal, given the column name.
GetProviderSpecificFieldType (inherited from DbDataReader)	Returns the provider-specific field type of the specified column.
GetProviderSpecificValue (inherited from DbDataReader)	Gets the value of the specified column as an instance of Object.

Member name	Description
GetProviderSpecificValues (inherited from DbDataReader)	Gets all provider-specific attribute columns in the collection for the current row.
GetSchemaTable method	Returns a DataTable that describes the column metadata of the SADataReader.
GetString method	Returns the value of the specified column as a string.
GetTimeSpan method	Returns the value of the specified column as a TimeSpan object.
GetUInt16 method	Returns the value of the specified column as a 16-bit unsigned integer.
GetUInt32 method	Returns the value of the specified column as a 32-bit unsigned integer.
GetUInt64 method	Returns the value of the specified column as a 64-bit unsigned integer.
GetValue methods	Returns the value of the specified column as an Object.
GetValues method	Gets all the columns in the current row.
IsDBNull method	Returns a value indicating whether the column contains NULL values.
NextResult method	Advances the SADataReader to the next result, when reading the results of batch SQL statements.
Read method	Reads the next row of the result set and moves the SADataReader to that row.
myDispose method	Frees the resources associated with the object.

See also

- "SADataReader class" on page 289
- "ExecuteReader() method" on page 211

Depth property

Gets a value indicating the depth of nesting for the current row. The outermost table has a depth of zero.

Syntax

Visual Basic

```
Public Overrides Readonly Property Depth As Integer
```

C#

```
public override int Depth  { get;}
```


Property value

The depth of nesting for the current row.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

FieldCount property

Gets the number of columns in the result set.

Syntax

Visual Basic

```
Public Overrides Readonly Property FieldCount As Integer
```

C#

```
public override int FieldCount  { get;}
```

Property value

The number of columns in the current record.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

HasRows property

Gets a value that indicates whether the SADataReader contains one or more rows.

Syntax

Visual Basic

```
Public Overrides Readonly Property HasRows As Boolean
```

C#

```
public override bool HasRows  { get;}
```

Property value

True if the SADataReader contains one or more rows; otherwise, false.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

IsClosed property

Gets a values that indicates whether the SADataReader is closed.

Syntax

Visual Basic

```
Public Overrides Readonly Property IsClosed As Boolean
```

C#

```
public override bool IsClosed  { get;}
```

Property value

True if the SADataReader is closed; otherwise, false.

Remarks

IsClosed and RecordsAffected are the only properties that you can call after the SADataReader is closed.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

Item properties

Returns the value of a column in its native format. In C#, this property is the indexer for the SADataReader class.

Item(Int32) property

Returns the value of a column in its native format. In C#, this property is the indexer for the SADataReader class.

Syntax

Visual Basic

```
Public Overrides Default Readonly Property Item ( _
   ByVal index As Integer _
) As Object
```

C#

```
public override object this [
   int index
] { get;}
```


Parameters

- **index** The column ordinal.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "Item properties" on page 294

Item(String) property

Returns the value of a column in its native format. In C#, this property is the indexer for the SADataReader class.

Syntax

Visual Basic

```
Public Overrides Default Readonly Property Item ( _
   ByVal name As String _
) As Object
```

C#

```
public override object this [
   string  name
] { get;}
```

Parameters

- **name** The column name.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "Item properties" on page 294

RecordsAffected property

The number of rows changed, inserted, or deleted by execution of the SQL statement.

Syntax

Visual Basic

```
Public Overrides Readonly Property RecordsAffected As Integer
```

C#

```
public override int RecordsAffected  { get;}
```

Property value

The number of rows changed, inserted, or deleted. This is 0 if no rows were affected or the statement failed, or -1 for SELECT statements.

Remarks

The number of rows changed, inserted, or deleted. The value is 0 if no rows were affected or the statement failed, and -1 for SELECT statements.

The value of this property is cumulative. For example, if two records are inserted in batch mode, the value of RecordsAffected will be two.

IsClosed and RecordsAffected are the only properties that you can call after the SADataReader is closed.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

Close method

Closes the SADataReader.

Syntax

Visual Basic

Public Overrides Sub **Close()**

C#

public override void **Close();**

Remarks

You must explicitly call the Close method when you are finished using the SADataReader.

When running in autocommit mode, a COMMIT is issued as a side effect of closing the SADataReader.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetBoolean method

Returns the value of the specified column as a Boolean.

Syntax

Visual Basic

Public Overrides Function **GetBoolean(** _


```
  ByVal ordinal As Integer _
) As Boolean
```

C#

```
public override bool GetBoolean(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the column.

Remarks

No conversions are performed, so the data retrieved must already be a Boolean.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "GetOrdinal method" on page 309
- "GetFieldType method" on page 304

GetByte method

Returns the value of the specified column as a Byte.

Syntax

Visual Basic

```
Public Overrides Function GetByte( _
  ByVal ordinal As Integer _
) As Byte
```

C#

```
public override byte GetByte(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the column.

Remarks

No conversions are performed, so the data retrieved must already be a byte.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetBytes method

Reads a stream of bytes from the specified column offset into the buffer as an array, starting at the given buffer offset.

Syntax

Visual Basic

```
Public Overrides Function GetBytes( _
   ByVal ordinal As Integer, _
   ByVal dataIndex As Long, _
   ByVal buffer As Byte(), _
   ByVal bufferIndex As Integer, _
   ByVal length As Integer _
) As Long
```

C#

```
public override long GetBytes(
   int ordinal,
   long dataIndex,
   byte[] buffer,
   int bufferIndex,
   int length
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
- **dataIndex** The index within the column value from which to read bytes.
- **buffer** An array in which to store the data.
- **bufferIndex** The index in the array to start copying data.
- **length** The maximum length to copy into the specified buffer.

Return value

The number of bytes read.

Remarks

GetBytes returns the number of available bytes in the field. In most cases this is the exact length of the field. However, the number returned may be less than the true length of the field if GetBytes has already been

used to obtain bytes from the field. This may be the case, for example, when the SADataReader is reading a large data structure into a buffer.

If you pass a buffer that is a null reference (Nothing in Visual Basic), GetBytes returns the length of the field in bytes.

No conversions are performed, so the data retrieved must already be a byte array.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetChar method

Returns the value of the specified column as a character.

Syntax

Visual Basic

```
Public Overrides Function GetChar( _
  ByVal ordinal As Integer _
) As Char
```

C#

```
public override char GetChar(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the column.

Remarks

No conversions are performed, so the data retrieved must already be a character.

Call the SADataReader.IsDBNull method to check for null values before calling this method.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "IsDBNull method" on page 317
- "IsDBNull method" on page 317

GetChars method

Reads a stream of characters from the specified column offset into the buffer as an array starting at the given buffer offset.

Syntax

Visual Basic

```
Public Overrides Function GetChars( _
   ByVal ordinal As Integer, _
   ByVal dataIndex As Long, _
   ByVal buffer As Char(), _
   ByVal bufferIndex As Integer, _
   ByVal length As Integer _
) As Long
```

C#

```
public override long GetChars(
   int ordinal,
   long dataIndex,
   char[] buffer,
   int bufferIndex,
   int length
);
```

Parameters

- **ordinal** The zero-based column ordinal.
- **dataIndex** The index within the row from which to begin the read operation.
- **buffer** The buffer into which to copy data.
- **bufferIndex** The index for buffer to begin the read operation.
- **length** The number of characters to read.

Return value

The actual number of characters read.

Remarks

GetChars returns the number of available characters in the field. In most cases this is the exact length of the field. However, the number returned may be less than the true length of the field if GetChars has already been used to obtain characters from the field. This may be the case, for example, when the SADataReader is reading a large data structure into a buffer.

If you pass a buffer that is a null reference (Nothing in Visual Basic), GetChars returns the length of the field in characters.

No conversions are performed, so the data retrieved must already be a character array.

For information about handling BLOBs, see "Handling BLOBs" on page 130.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetData method

This method is not supported. When called, it throws an InvalidOperationException.

Syntax

Visual Basic

```
Public Function GetData( _
   ByVal i As Integer _
) As IDataReader
```

C#

```
public IDataReader GetData(
   int i
);
```

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- InvalidOperationException

GetDataTypeName method

Returns the name of the source data type.

Syntax

Visual Basic

```
Public Overrides Function GetDataTypeName( _
   ByVal index As Integer _
) As String
```

C#

```
public override string  GetDataTypeName(
   int index
);
```

Parameters

- **index** The zero-based column ordinal.

Return value

The name of the back-end data type.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetDateTime method

Returns the value of the specified column as a DateTime object.

Syntax

Visual Basic

```
Public Overrides Function GetDateTime( _
   ByVal ordinal As Integer _
) As Date
```

C#

```
public override DateTime GetDateTime(
   int ordinal
);
```

Parameters

- **ordinal** The zero-based column ordinal.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a DateTime object.

Call the SADataReader.IsDBNull method to check for null values before calling this method.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "IsDBNull method" on page 317

GetDecimal method

Returns the value of the specified column as a Decimal object.

Syntax

Visual Basic

```
Public Overrides Function GetDecimal( _
   ByVal ordinal As Integer _
) As Decimal
```


C#

```
public override decimal GetDecimal(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a Decimal object.

Call the SADataReader.IsDBNull method to check for null values before calling this method.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "IsDBNull method" on page 317

GetDouble method

Returns the value of the specified column as a double-precision floating point number.

Syntax

Visual Basic

```
Public Overrides Function GetDouble( _
  ByVal ordinal As Integer _
) As Double
```

C#

```
public override double GetDouble(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a double-precision floating point number.

Call the SADataReader.IsDBNull method to check for null values before calling this method.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "IsDBNull method" on page 317

GetEnumerator method

Returns a IEnumerator that iterates through the SADataReader object.

Syntax

Visual Basic

```
Public Overrides Function GetEnumerator() As IEnumerator
```

C#

```
public override IEnumerator GetEnumerator();
```

Return value

A IEnumerator for the SADataReader object.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "SADataReader class" on page 289

GetFieldType method

Returns the Type that is the data type of the object.

Syntax

Visual Basic

```
Public Overrides Function GetFieldType( _
  ByVal index As Integer _
) As Type
```

C#

```
public override Type GetFieldType(
  int index
);
```


Parameters

- **index** The zero-based column ordinal.

Return value

The type that is the data type of the object.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetFloat method

Returns the value of the specified column as a single-precision floating point number.

Syntax

Visual Basic

```
Public Overrides Function GetFloat( _
  ByVal ordinal As Integer _
) As Single
```

C#

```
public override float  GetFloat(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a single-precision floating point number.

Call the SADataReader.IsDBNull method to check for null values before calling this method.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "IsDBNull method" on page 317

GetGuid method

Returns the value of the specified column as a global unique identifier (GUID).

Syntax

Visual Basic

```
Public Overrides Function GetGuid( _
   ByVal ordinal As Integer _
) As Guid
```

C#

```
public override Guid GetGuid(
   int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

The data retrieved must already be a globally-unique identifier or binary(16).

Call the SADataReader.IsDBNull method to check for null values before calling this method.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "IsDBNull method" on page 317

GetInt16 method

Returns the value of the specified column as a 16-bit signed integer.

Syntax

Visual Basic

```
Public Overrides Function GetInt16( _
   ByVal ordinal As Integer _
) As Short
```

C#

```
public override short GetInt16(
```



```
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a 16-bit signed integer.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetInt32 method

Returns the value of the specified column as a 32-bit signed integer.

Syntax

Visual Basic

```
Public Overrides Function GetInt32( _
  ByVal ordinal As Integer _
) As Integer
```

C#

```
public override int GetInt32(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a 32-bit signed integer.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetInt64 method

Returns the value of the specified column as a 64-bit signed integer.

Syntax

Visual Basic

```
Public Overrides Function GetInt64( _
   ByVal ordinal As Integer _
) As Long
```

C#

```
public override long GetInt64(
   int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a 64-bit signed integer.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetName method

Returns the name of the specified column.

Syntax

Visual Basic

```
Public Overrides Function GetName( _
   ByVal index As Integer _
) As String
```

C#

```
public override string  GetName(
   int index
);
```


Parameters

- **index** The zero-based index of the column.

Return value

The name of the specified column.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetOrdinal method

Returns the column ordinal, given the column name.

Syntax

Visual Basic

```
Public Overrides Function GetOrdinal( _
   ByVal name As String _
) As Integer
```

C#

```
public override int GetOrdinal(
   string  name
);
```

Parameters

- **name** The column name.

Return value

The zero-based column ordinal.

Remarks

GetOrdinal performs a case-sensitive lookup first. If it fails, a second case-insensitive search is made.

GetOrdinal is Japanese kana-width insensitive.

Because ordinal-based lookups are more efficient than named lookups, it is inefficient to call GetOrdinal within a loop. You can save time by calling GetOrdinal once and assigning the results to an integer variable for use within the loop.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetSchemaTable method

Returns a DataTable that describes the column metadata of the SADataReader.

Syntax

Visual Basic

Public Overrides Function **GetSchemaTable()** As DataTable

C#

public override DataTable **GetSchemaTable();**

Return value

A DataTable that describes the column metadata.

Remarks

This method returns metadata about each column in the following order:

DataTable column	Description
ColumnName	The name of the column or a null reference (Nothing in Visual Basic) if the column has no name. If the column is aliased in the SQL query, the alias is returned. Note that in result sets, not all columns have names and not all column names are unique.
ColumnOrdinal	The ID of the column. The value is in the range [0, FieldCount -1].
ColumnSize	For sized columns, the maximum length of a value in the column. For other columns, this is the size in bytes of the data type.
NumericPrecision	The precision of a numeric column or DBNull if the column is not numeric.
NumericScale	The scale of a numeric column or DBNull if the column is not numeric.
IsUnique	True if the column is a non-computed unique column in the table (BaseTableName) it is taken from.
IsKey	True if the column is one of a set of columns in the result set that taken together from a unique key for the result set. The set of columns with IsKey set to true does not need to be the minimal set that uniquely identifies a row in the result set.
BaseServerName	The name of the SQL Anywhere database server used by the SADataReader.

DataTable column	Description
BaseCatalogName	The name of the catalog in the database that contains the column. This value is always DBNull.
BaseColumnName	The original name of the column in the table BaseTableName of the database or DBNull if the column is computed or if this information cannot be determined.
BaseSchemaName	The name of the schema in the database that contains the column.
BaseTableName	The name of the table in the database that contains the column, or DBNull if column is computed or if this information cannot be determined.
DataType	The .NET data type that is most appropriate for this type of column.
AllowDBNull	True if the column is nullable, false if the column is not nullable or if this information cannot be determined.
ProviderType	The type of the column.
IsAliased	True if the column name is an alias, false if it is not an alias.
IsExpression	True if the column is an expression, false if it is a column value.
IsIdentity	True if the column is an identity column, false if it is not an identity column.
IsAutoIncrement	True if the column is an autoincrement or global autoincrement column, false otherwise (or if this information cannot be determined).
IsRowVersion	True if the column contains a persistent row identifier that cannot be written to, and has no meaningful value except to identify the row.
IsHidden	True if the column is hidden, false otherwise.
IsLong	True if the column is a long varchar, long nvarchar, or a long binary column, false otherwise.
IsReadOnly	True if the column is read-only, false if the column is modifiable or if its access cannot be determined.

For more information about these columns, see the .NET Framework documentation for SqlDataReader.GetSchemaTable.

For more information, see "Obtaining DataReader schema information" on page 120.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetString method

Returns the value of the specified column as a string.

Syntax

Visual Basic

```
Public Overrides Function GetString( _
  ByVal ordinal As Integer _
) As String
```

C#

```
public override string  GetString(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a string.

Call the SADataReader.IsDBNull method to check for NULL values before calling this method.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "IsDBNull method" on page 317

GetTimeSpan method

Returns the value of the specified column as a TimeSpan object.

Syntax

Visual Basic

```
Public Function GetTimeSpan( _
```



```
  ByVal ordinal As Integer _
) As TimeSpan
```

C#

```
public TimeSpan GetTimeSpan(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

The column must be a SQL Anywhere TIME data type. The data is converted to TimeSpan. The Days property of TimeSpan is always set to 0.

Call SADataReader.IsDBNull method to check for NULL values before calling this method.

For more information, see "Obtaining time values" on page 130.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "IsDBNull method" on page 317

GetUInt16 method

Returns the value of the specified column as a 16-bit unsigned integer.

Syntax

Visual Basic

```
Public Function GetUInt16( _
  ByVal ordinal As Integer _
) As UInt16
```

C#

```
public ushort GetUInt16(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a 16-bit unsigned integer.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetUInt32 method

Returns the value of the specified column as a 32-bit unsigned integer.

Syntax

Visual Basic

```
Public Function GetUInt32( _
  ByVal ordinal As Integer _
) As UInt32
```

C#

```
public uint GetUInt32(
  int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a 32-bit unsigned integer.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetUInt64 method

Returns the value of the specified column as a 64-bit unsigned integer.

Syntax

Visual Basic

```
Public Function GetUInt64( _
   ByVal ordinal As Integer _
) As UInt64
```

C#

```
public ulong GetUInt64(
   int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column.

Remarks

No conversions are performed, so the data retrieved must already be a 64-bit unsigned integer.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

GetValue methods

Returns the value of the specified column as an Object.

GetValue(Int32) method

Returns the value of the specified column as an Object.

Syntax

Visual Basic

```
Public Overrides Function GetValue( _
   ByVal ordinal As Integer _
) As Object
```

C#

```
public override object GetValue(
   int ordinal
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return value

The value of the specified column as an object.

Remarks

This method returns DBNull for NULL database columns.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "GetValue methods" on page 315

GetValue(Int32, Int64, Int32) method

Returns a substring of the value of the specified column as an Object.

Syntax

Visual Basic

```
Public Function GetValue( _
   ByVal ordinal As Integer, _
   ByVal index As Long, _
   ByVal length As Integer _
) As Object
```

C#

```
public object GetValue(
   int ordinal,
   long index,
   int length
);
```

Parameters

- **ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
- **index** A zero-based index of the substring of the value to be obtained.
- **length** The length of the substring of the value to be obtained.

Return value

The substring value is returned as an object.

Remarks

This method returns DBNull for NULL database columns.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290
- "GetValue methods" on page 315

GetValues method

Gets all the columns in the current row.

Syntax

Visual Basic

```
Public Overrides Function GetValues( _
   ByVal values As Object() _
) As Integer
```

C#

```
public override int GetValues(
   object[] values
);
```

Parameters

- **values** An array of objects that holds an entire row of the result set.

Return value

The number of objects in the array.

Remarks

For most applications, the GetValues method provides an efficient means for retrieving all columns, rather than retrieving each column individually.

You can pass an Object array that contains fewer than the number of columns contained in the resulting row. Only the amount of data the Object array holds is copied to the array. You can also pass an Object array whose length is more than the number of columns contained in the resulting row.

This method returns DBNull for NULL database columns.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

IsDBNull method

Returns a value indicating whether the column contains NULL values.

Syntax

Visual Basic

```
Public Overrides Function IsDBNull( _
   ByVal ordinal As Integer _
) As Boolean
```

C#

```
public override bool IsDBNull(
   int ordinal
);
```

Parameters

- **ordinal** The zero-based column ordinal.

Return value

Returns true if the specified column value is equivalent to DBNull. Otherwise, it returns false.

Remarks

Call this method to check for NULL column values before calling the typed get methods (for example, GetByte, GetChar, and so on) to avoid raising an exception.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

NextResult method

Advances the SADataReader to the next result, when reading the results of batch SQL statements.

Syntax

Visual Basic

```
Public Overrides Function NextResult() As Boolean
```

C#

```
public override bool NextResult();
```

Return value

Returns true if there are more result sets. Otherwise, it returns false.

Remarks

Used to process multiple results, which can be generated by executing batch SQL statements.

By default, the data reader is positioned on the first result.

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

Read method

Reads the next row of the result set and moves the SADataReader to that row.

Syntax

Visual Basic

Public Overrides Function **Read()** As Boolean

C#

public override bool **Read();**

Return value

Returns true if there are more rows. Otherwise, it returns false.

Remarks

The default position of the SADataReader is prior to the first record. Therefore, you must call Read to begin accessing any data.

Example

The following code fills a listbox with the values in a single column of results.

```
while( reader.Read() )
{
    listResults.Items.Add(
        reader.GetValue( 0 ).ToString() );
}
listResults.EndUpdate();
reader.Close();
```

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

myDispose method

Frees the resources associated with the object.

Syntax

Visual Basic

Public Sub **myDispose()**

C#

public void **myDispose();**

See also

- "SADataReader class" on page 289
- "SADataReader members" on page 290

SADataSourceEnumerator class

Provides a mechanism for enumerating all available instances of SQL Anywhere database servers within the local network. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SADataSourceEnumerator
  Inherits DbDataSourceEnumerator
```

C#

```
public sealed class SADataSourceEnumerator : DbDataSourceEnumerator
```

Remarks

There is no constructor for SADataSourceEnumerator.

The SADataSourceEnumerator class is not available in the .NET Compact Framework 2.0.

See also

- "SADataSourceEnumerator members" on page 320

SADataSourceEnumerator members

Public properties

Member name	Description
Instance property	Gets an instance of SADataSourceEnumberator, which can be used to retrieve information about all visible SQL Anywhere database servers.

Public methods

Member name	Description
GetDataSources method	Retrieves a DataTable containing information about all visible SQL Anywhere database servers.

See also

- "SADataSourceEnumerator class" on page 320

Instance property

Gets an instance of SADataSourceEnumberator, which can be used to retrieve information about all visible SQL Anywhere database servers.

Syntax

Visual Basic

Public Shared Readonly Property **Instance** As SADataSourceEnumerator

C#

public const SADataSourceEnumerator **Instance** { get;}

See also

- "SADataSourceEnumerator class" on page 320
- "SADataSourceEnumerator members" on page 320

GetDataSources method

Retrieves a DataTable containing information about all visible SQL Anywhere database servers.

Syntax

Visual Basic

Public Overrides Function **GetDataSources()** As DataTable

C#

public override DataTable **GetDataSources();**

Remarks

The returned table has four columns: ServerName, IPAddress, PortNumber, and DataBaseNames. There is a row in the table for each available database server.

Example

The following code fills a DataTable with information for each database server that is available.

```
DataTable servers = SADataSourceEnumerator.Instance.GetDataSources();
```

See also

- "SADataSourceEnumerator class" on page 320
- "SADataSourceEnumerator members" on page 320

SADbType enumeration

Enumerates the SQL Anywhere .NET database data types.

Syntax

Visual Basic

Public Enum **SADbType**

C#

public enum **SADbType**

Remarks

The table below lists which .NET types are compatible with each SADbType. In the case of integral types, table columns can always be set using smaller integer types, but can also be set using larger types as long as the actual value is within the range of the type.

SADbType	Compatible .NET type	C# built-in type	Visual Basic built-in type
BigInt	System.Int64	long	Long
Binary, **VarBinary**	System.Byte[], or System.Guid if size is 16	byte[]	Byte()
Bit	System.Boolean	bool	Boolean
Char, **VarChar**	System.String	String	String
Date	System.DateTime	DateTime (no built-in type)	Date
DateTime, **TimeStamp**	System.DateTime	DateTime (no built-in type)	Date
Decimal, **Numeric**	System.String	decimal	Decimal
Double	System.Double	double	Double
Float, **Real**	System.Single	float	Single
Image	System.Byte[]	byte[]	Byte()
Integer	System.Int32	int	Integer
LongBinary	System.Byte[]	byte[]	Byte()

SADbType	Compatible .NET type	C# built-in type	Visual Basic built-in type
LongNVarChar	System.String	String	String
LongVarChar	System.String	String	String
Money	System.String	decimal	Decimal
NChar	System.String	String	String
NText	System.String	String	String
Numeric	System.String	decimal	Decimal
NVarChar	System.String	String	String
SmallDateTime	System.DateTime	DateTime (no built-in type)	Date
SmallInt	System.Int16	short	Short
SmallMoney	System.String	decimal	Decimal
SysName	System.String	String	String
Text	System.String	String	String
Time	System.TimeSpan	TimeSpan (no built-in type)	TimeSpan (no built-in type)
TimeStamp	System.DateTime	DateTime (no built-in type)	Date
TinyInt	System.Byte	byte	Byte
UniqueIdentifier	System.Guid	Guid (no built-in type)	Guid (no built-in type)
UniqueIdentifierStr	System.String	String	String
UnsignedBigInt	System.UInt64	ulong	UInt64 (no built-in type)
UnsignedInt	System.UInt32	uint	UInt64 (no built-in type)
UnsignedSmallInt	System.UInt16	ushort	UInt64 (no built-in type)

SADbType	Compatible .NET type	C# built-in type	Visual Basic built-in type
Xml	System.Xml	String	String

Binary columns of length 16 are fully compatible with the UniqueIdentifier type.

Members

Member name	Description	Value
BigInt	Signed 64-bit integer.	1
Binary	Binary data, with a specified maximum length. The enumeration values Binary and VarBinary are aliases of each other.	2
Bit	1-bit flag.	3
Char	Character data, with a specified length. This type always supports Unicode characters. The types Char and VarChar are fully compatible.	4
Date	Date information.	5
DateTime	Timestamp information (date, time). The enumeration values DateTime and TimeStamp are aliases of each other.	6
Decimal	Exact numerical data, with a specified precision and scale. The enumeration values Decimal and Numeric are aliases of each other.	7
Double	Double precision floating-point number (8 bytes).	8
Float	Single precision floating-point number (4 bytes). The enumeration values Float and Real are aliases of each other.	9
Image	Stores binary data of arbitrary length.	10
Integer	Unsigned 32-bit integer.	11
LongBinary	Binary data, with variable length.	12
LongNVarchar	Character data in the NCHAR character set, with variable length. This type always supports Unicode characters.	13

Member name	Description	Value
LongVarbit	Bit arrays, with variable length.	14
LongVarchar	Character data, with variable length. This type always supports Unicode characters.	15
Money	Monetary data.	16
NChar	Stores Unicode character data, up to 8191 characters.	17
NText	Stores Unicode character data of arbitrary length.	18
Numeric	Exact numerical data, with a specified precision and scale. The enumeration values Decimal and Numeric are aliases of each other.	19
NVarChar	Stores Unicode character data, up to 8191 characters.	20
Real	Single precision floating-point number (4 bytes). The enumeration values Float and Real are aliases of each other.	21
SmallDateTime	A domain, implemented as TIMESTAMP.	22
SmallInt	Signed 16-bit integer.	23
SmallMoney	Stores monetary data that is less than one million currency units.	24
SysName	Stores character data of arbitrary length.	25
Text	Stores character data of arbitrary length.	26
Time	Time information.	27
TimeStamp	Timestamp information (date, time). The enumeration values DateTime and TimeStamp are aliases of each other.	28
TinyInt	Unsigned 8-bit integer.	29
UniqueIdentifier	Universally Unique Identifier (UUID/GUID).	30

Member name	Description	Value
UniqueIdentifierStr	A domain, implemented as CHAR(36). UniqueIdentifierStr is used for remote data access when mapping Microsoft SQL Server uniqueidentifier columns.	31
UnsignedBigInt	Unsigned 64-bit integer.	32
UnsignedInt	Unsigned 32-bit integer.	33
UnsignedSmallInt	Unsigned 16-bit integer.	34
VarBinary	Binary data, with a specified maximum length. The enumeration values **Binary** and **VarBinary** are aliases of each other.	35
VarBit	Bit arrays that are from 1 to 32767 bits in length.	36
VarChar	Character data, with a specified maximum length. This type always supports Unicode characters. The types Char and VarChar are fully compatible.	37
Xml	XML data. This type stores character data of arbitrary length, and is used to store XML documents.	38

See also

- "GetFieldType method" on page 304
- "GetDataTypeName method" on page 301

SADefault class

Represents a parameter with a default value. This is a static class and so cannot be inherited or instantiated.

Syntax

Visual Basic

Public NotInheritable Class **SADefault**

C#

public sealed class **SADefault**

Remarks

There is no constructor for SADefault.

```
SAParameter parm = new SAParameter();
parm.Value = SADefault.Value;
```


See also

- "SADefault members" on page 327

SADefault members

Public fields

Member name	Description
Value field	Gets the value for a default parameter. This field is read-only and static. This field is read-only.

See also

- "SADefault class" on page 326

Value field

Gets the value for a default parameter. This field is read-only and static. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **Value** As SADefault

C#

public const SADefault **Value** ;

See also

- "SADefault class" on page 326
- "SADefault members" on page 327

SAError class

Collects information relevant to a warning or error returned by the data source. This class cannot be inherited.

Syntax

Visual Basic

Public NotInheritable Class **SAError**

C#

public sealed class **SAError**

Remarks

There is no constructor for SAError.

For information about error handling, see "Error handling and the SQL Anywhere .NET Data Provider" on page 136.

See also

- "SAError members" on page 328

SAError members

Public properties

Member name	Description
Message property	Returns a short description of the error.
NativeError property	Returns database-specific error information.
Source property	Returns the name of the provider that generated the error.
SqlState property	The SQL Anywhere five-character SQLSTATE following the ANSI SQL standard.

Public methods

Member name	Description
ToString method	The complete text of the error message.

See also

- "SAError class" on page 327

Message property

Returns a short description of the error.

Syntax

Visual Basic

```
Public Readonly Property Message As String
```

C#

```
public string  Message  { get;}
```


See also

- "SAError class" on page 327
- "SAError members" on page 328

NativeError property

Returns database-specific error information.

Syntax

Visual Basic

Public Readonly Property **NativeError** As Integer

C#

public int **NativeError** { get;}

See also

- "SAError class" on page 327
- "SAError members" on page 328

Source property

Returns the name of the provider that generated the error.

Syntax

Visual Basic

Public Readonly Property **Source** As String

C#

public string **Source** { get;}

See also

- "SAError class" on page 327
- "SAError members" on page 328

SqlState property

The SQL Anywhere five-character SQLSTATE following the ANSI SQL standard.

Syntax

Visual Basic

Public Readonly Property **SqlState** As String

C#

public string **SqlState** { get;}

See also

- "SAError class" on page 327
- "SAError members" on page 328

ToString method

The complete text of the error message.

Syntax

Visual Basic

Public Overrides Function **ToString()** As String

C#

public override string **ToString();**

Example

The return value is a string is in the form **SAError:**, followed by the Message. For example:

```
SAError:UserId or Password not valid.
```

See also

- "SAError class" on page 327
- "SAError members" on page 328

SAErrorCollection class

Collects all errors generated by the SQL Anywhere .NET Data Provider. This class cannot be inherited.

Syntax

Visual Basic

Public NotInheritable Class **SAErrorCollection**
  Implements ICollection, IEnumerable

C#

public sealed class **SAErrorCollection** : ICollection, IEnumerable

Remarks

There is no constructor for SAErrorCollection. Typically, an SAErrorCollection is obtained from the SAException.Errors property.

Implements:ICollection, IEnumerable

For information about error handling, see "Error handling and the SQL Anywhere .NET Data Provider" on page 136.

See also

- "SAErrorCollection members" on page 331
- "Errors property" on page 335
- SqlClientFactory.CanCreateDataSourceEnumerator

SAErrorCollection members

Public properties

Member name	Description
Count property	Returns the number of errors in the collection.
Item property	Returns the error at the specified index.

Public methods

Member name	Description
CopyTo method	Copies the elements of the SAErrorCollection into an array, starting at the given index within the array.
GetEnumerator method	Returns an enumerator that iterates through the SAErrorCollection.

See also

- "SAErrorCollection class" on page 330
- "Errors property" on page 335
- SqlClientFactory.CanCreateDataSourceEnumerator

Count property

Returns the number of errors in the collection.

Syntax

Visual Basic

```
NotOverridable Public Readonly Property Count As Integer
```

C#

```
public int Count  { get;}
```

See also

- "SAErrorCollection class" on page 330
- "SAErrorCollection members" on page 331

Item property

Returns the error at the specified index.

Syntax

Visual Basic

```
Public Readonly Property Item ( _
   ByVal index As Integer _
) As SAError
```

C#

```
public SAError this [
   int index
] { get;}
```

Parameters

- **index** The zero-based index of the error to retrieve.

Property value

An SAError object that contains the error at the specified index.

See also

- "SAErrorCollection class" on page 330
- "SAErrorCollection members" on page 331
- "SAError class" on page 327

CopyTo method

Copies the elements of the SAErrorCollection into an array, starting at the given index within the array.

Syntax

Visual Basic

```
NotOverridable Public Sub CopyTo( _
   ByVal array As Array, _
   ByVal index As Integer _
)
```

C#

```
public void CopyTo(
   Array array,
```


int *index*
);

Parameters

- **array** The array into which to copy the elements.
- **index** The starting index of the array.

See also

- "SAErrorCollection class" on page 330
- "SAErrorCollection members" on page 331

GetEnumerator method

Returns an enumerator that iterates through the SAErrorCollection.

Syntax

Visual Basic

NotOverridable Public Function **GetEnumerator()** As IEnumerator

C#

public IEnumerator **GetEnumerator();**

Return value

An IEnumerator for the SAErrorCollection.

See also

- "SAErrorCollection class" on page 330
- "SAErrorCollection members" on page 331

SAException class

The exception that is thrown when SQL Anywhere returns a warning or error.

Syntax

Visual Basic

Public Class **SAException**
  Inherits DbException

C#

public class **SAException** : DbException

Remarks

There is no constructor for SAException. Typically, an SAException object is declared in a catch. For example:

```
...
catch( SAException ex )
{
    MessageBox.Show( ex.Errors[0].Message, "Error" );
}
```

For information about error handling, see "Error handling and the SQL Anywhere .NET Data Provider" on page 136.

See also

- "SAException members" on page 334

SAException members

Public properties

Member name	Description
Data (inherited from Exception)	
ErrorCode (inherited from ExternalException)	
Errors property	Returns a collection of one or more "SAError class" on page 327 objects.
HelpLink (inherited from Exception)	
InnerException (inherited from Exception)	
Message property	Returns the text describing the error.
NativeError property	Returns database-specific error information.
Source property	Returns the name of the provider that generated the error.
StackTrace (inherited from Exception)	
TargetSite (inherited from Exception)	

Public methods

Member name	Description
GetBaseException (inherited from Exception)	
GetObjectData method	Sets the SerializationInfo with information about the exception. Overrides Exception.GetObjectData.
GetType (inherited from Exception)	
ToString (inherited from Exception)	

See also

- "SAException class" on page 333

Errors property

Returns a collection of one or more "SAError class" on page 327 objects.

Syntax

Visual Basic

```
Public Readonly Property Errors As SAErrorCollection
```

C#

```
public SAErrorCollection Errors  { get;}
```

Remarks

The SAErrorCollection object always contains at least one instance of the SAError object.

See also

- "SAException class" on page 333
- "SAException members" on page 334
- "SAErrorCollection class" on page 330
- "SAError class" on page 327

Message property

Returns the text describing the error.

Syntax

Visual Basic

Public Overrides Readonly Property **Message** As String

C#

public override string **Message** { get;}

Remarks

This method returns a single string that contains a concatenation of all of the Message properties of all of the SAError objects in the Errors collection. Each message, except the last one, is followed by a carriage return.

See also

- "SAException class" on page 333
- "SAException members" on page 334
- "SAError class" on page 327

NativeError property

Returns database-specific error information.

Syntax

Visual Basic

Public Readonly Property **NativeError** As Integer

C#

public int **NativeError** { get;}

See also

- "SAException class" on page 333
- "SAException members" on page 334

Source property

Returns the name of the provider that generated the error.

Syntax

Visual Basic

Public Overrides Readonly Property **Source** As String

C#

public override string **Source** { get;}

See also

- "SAException class" on page 333
- "SAException members" on page 334

GetObjectData method

Sets the SerializationInfo with information about the exception. Overrides Exception.GetObjectData.

Syntax

Visual Basic

```
Public Overrides Sub GetObjectData( _
  ByVal info As SerializationInfo, _
  ByVal context As StreamingContext _
)
```

C#

```
public override void GetObjectData(
  SerializationInfo info,
  StreamingContext context
);
```

Parameters

- **info** The SerializationInfo that holds the serialized object data about the exception being thrown.
- **context** The StreamingContext that contains contextual information about the source or destination.

See also

- "SAException class" on page 333
- "SAException members" on page 334

SAFactory class

Represents a set of methods for creating instances of the iAnywhere.Data.SQLAnywhere provider's implementation of the data source classes. This is a static class and so cannot be inherited or instantiated.

Syntax

Visual Basic

```
Public NotInheritable Class SAFactory
  Inherits DbProviderFactory
  Implements IServiceProvider
```

C#

```
public sealed class SAFactory : DbProviderFactory,
  IServiceProvider
```

Remarks

There is no constructor for SAFactory.

ADO.NET 2.0 adds two new classes, DbProviderFactories and DbProviderFactory, to make provider independent code easier to write. To use them with SQL Anywhere specify iAnywhere.Data.SQLAnywhere as the provider invariant name passed to GetFactory. For example:

```
' Visual Basic
Dim factory As DbProviderFactory = _
  DbProviderFactories.GetFactory( "iAnywhere.Data.SQLAnywhere" )
Dim conn As DbConnection = _
  factory.CreateConnection()
// C#
DbProviderFactory factory =
        DbProviderFactories.GetFactory("iAnywhere.Data.SQLAnywhere" );
DbConnection conn = factory.CreateConnection();
```

In this example, conn is created as an SAConnection object.

For an explanation of provider factories and generic programming in ADO.NET 2.0, see http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/vsgenerics.asp.

Restrictions: The SAFactory class is not available in the .NET Compact Framework 2.0.

Inherits: DbProviderFactory

See also

- "SAFactory members" on page 338

SAFactory members

Public fields

Member name	Description
Instance field	Represents the singleton instance of the SAFactory class. This field is read-only.

Public properties

Member name	Description
CanCreateDataSourceEnumerator property	Always returns true, which indicates that an SADataSourceEnumerator object can be created.

Public methods

Member name	Description
CreateCommand method	Returns a strongly typed DbCommand instance.

Member name	Description
CreateCommandBuilder method	Returns a strongly typed DbCommandBuilder instance.
CreateConnection method	Returns a strongly typed DbConnection instance.
CreateConnectionStringBuilder method	Returns a strongly typed DbConnectionStringBuilder instance.
CreateDataAdapter method	Returns a strongly typed DbDataAdapter instance.
CreateDataSourceEnumerator method	Returns a strongly typed DbDataSourceEnumerator instance.
CreateParameter method	Returns a strongly typed DbParameter instance.
CreatePermission method	Returns a strongly-typed CodeAccessPermission instance.

See also

- "SAFactory class" on page 337

Instance field

Represents the singleton instance of the SAFactory class. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **Instance** As SAFactory

C#

public const SAFactory **Instance** ;

Remarks

SAFactory is a singleton class, which means only this instance of this class can exist.

Normally you would not use this field directly. Instead, you get a reference to this instance of SAFactory using DbProviderFactories.GetFactory. For an example, see the SAFactory description.

Restrictions: The SAFactory class is not available in the .NET Compact Framework 2.0.

See also

- "SAFactory class" on page 337
- "SAFactory members" on page 338
- "SAFactory class" on page 337

CanCreateDataSourceEnumerator property

Always returns true, which indicates that an SADataSourceEnumerator object can be created.

Syntax

Visual Basic

Public Overrides Readonly Property **CanCreateDataSourceEnumerator** As Boolean

C#

public override bool **CanCreateDataSourceEnumerator** { get;}

Property value

A new SACommand object typed as DbCommand.

See also

- "SAFactory class" on page 337
- "SAFactory members" on page 338
- "SADataSourceEnumerator class" on page 320
- "SACommand class" on page 192

CreateCommand method

Returns a strongly typed DbCommand instance.

Syntax

Visual Basic

Public Overrides Function **CreateCommand()** As DbCommand

C#

public override DbCommand **CreateCommand();**

Return value

A new SACommand object typed as DbCommand.

See also

- "SAFactory class" on page 337
- "SAFactory members" on page 338
- "SACommand class" on page 192

CreateCommandBuilder method

Returns a strongly typed DbCommandBuilder instance.

Syntax

Visual Basic

Public Overrides Function **CreateCommandBuilder()** As DbCommandBuilder

C#

public override DbCommandBuilder **CreateCommandBuilder();**

Return value

A new SACommand object typed as DbCommand.

See also

- "SAFactory class" on page 337
- "SAFactory members" on page 338
- "SACommand class" on page 192

CreateConnection method

Returns a strongly typed DbConnection instance.

Syntax

Visual Basic

Public Overrides Function **CreateConnection()** As DbConnection

C#

public override DbConnection **CreateConnection();**

Return value

A new SACommand object typed as DbCommand.

See also

- "SAFactory class" on page 337
- "SAFactory members" on page 338
- "SACommand class" on page 192

CreateConnectionStringBuilder method

Returns a strongly typed DbConnectionStringBuilder instance.

Syntax

Visual Basic

Public Overrides Function **CreateConnectionStringBuilder()** As DbConnectionStringBuilder

C#

public override DbConnectionstring Builder **CreateConnectionStringBuilder();**

Return value

A new SACommand object typed as DbCommand.

See also

- "SAFactory class" on page 337
- "SAFactory members" on page 338
- "SACommand class" on page 192

CreateDataAdapter method

Returns a strongly typed DbDataAdapter instance.

Syntax

Visual Basic

Public Overrides Function **CreateDataAdapter()** As DbDataAdapter

C#

public override DbDataAdapter **CreateDataAdapter();**

Return value

A new SACommand object typed as DbCommand.

See also

- "SAFactory class" on page 337
- "SAFactory members" on page 338
- "SACommand class" on page 192

CreateDataSourceEnumerator method

Returns a strongly typed DbDataSourceEnumerator instance.

Syntax

Visual Basic

Public Overrides Function **CreateDataSourceEnumerator()** As DbDataSourceEnumerator

C#

public override DbDataSourceEnumerator **CreateDataSourceEnumerator();**

Return value

A new SACommand object typed as DbCommand.

See also

- "SAFactory class" on page 337
- "SAFactory members" on page 338
- "SACommand class" on page 192

CreateParameter method

Returns a strongly typed DbParameter instance.

Syntax

Visual Basic

```
Public Overrides Function CreateParameter() As DbParameter
```

C#

```
public override DbParameter CreateParameter();
```

Return value

A new SACommand object typed as DbCommand.

See also

- "SAFactory class" on page 337
- "SAFactory members" on page 338
- "SACommand class" on page 192

CreatePermission method

Returns a strongly-typed CodeAccessPermission instance.

Syntax

Visual Basic

```
Public Overrides Function CreatePermission( _
   ByVal state As PermissionState _
) As CodeAccessPermission
```

C#

```
public override CodeAccessPermission CreatePermission(
   PermissionState state
);
```

Parameters

- **state** A member of the PermissionState enumeration.

Return value

A new SACommand object typed as DbCommand.

See also

- "SAFactory class" on page 337
- "SAFactory members" on page 338
- "SACommand class" on page 192

SAInfoMessageEventArgs class

Provides data for the InfoMessage event. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SAInfoMessageEventArgs
  Inherits EventArgs
```

C#

```
public sealed class SAInfoMessageEventArgs : EventArgs
```

Remarks

There is no constructor for SAInfoMessageEventArgs.

See also

- "SAInfoMessageEventArgs members" on page 344

SAInfoMessageEventArgs members

Public properties

Member name	Description
Errors property	Returns the collection of messages sent from the data source.
Message property	Returns the full text of the error sent from the data source.
MessageType property	Returns the type of the message. This can be one of: Action, Info, Status, or Warning.
NativeError property	Returns the SQL code returned by the database.

Member name	Description
Source property	Returns the name of the SQL Anywhere .NET Data Provider.

Public methods

Member name	Description
ToString method	Retrieves a string representation of the InfoMessage event.

See also

- "SAInfoMessageEventArgs class" on page 344

Errors property

Returns the collection of messages sent from the data source.

Syntax

Visual Basic

```
Public Readonly Property Errors As SAErrorCollection
```

C#

```
public SAErrorCollection Errors  { get;}
```

See also

- "SAInfoMessageEventArgs class" on page 344
- "SAInfoMessageEventArgs members" on page 344

Message property

Returns the full text of the error sent from the data source.

Syntax

Visual Basic

```
Public Readonly Property Message As String
```

C#

```
public string  Message  { get;}
```

See also

- "SAInfoMessageEventArgs class" on page 344
- "SAInfoMessageEventArgs members" on page 344

MessageType property

Returns the type of the message. This can be one of: Action, Info, Status, or Warning.

Syntax

Visual Basic

Public Readonly Property **MessageType** As SAMessageType

C#

public SAMessageType **MessageType** { get;}

See also

- "SAInfoMessageEventArgs class" on page 344
- "SAInfoMessageEventArgs members" on page 344

NativeError property

Returns the SQL code returned by the database.

Syntax

Visual Basic

Public Readonly Property **NativeError** As Integer

C#

public int **NativeError** { get;}

See also

- "SAInfoMessageEventArgs class" on page 344
- "SAInfoMessageEventArgs members" on page 344

Source property

Returns the name of the SQL Anywhere .NET Data Provider.

Syntax

Visual Basic

Public Readonly Property **Source** As String

C#

public string **Source** { get;}

See also

- "SAInfoMessageEventArgs class" on page 344
- "SAInfoMessageEventArgs members" on page 344

ToString method

Retrieves a string representation of the InfoMessage event.

Syntax

Visual Basic

```
Public Overrides Function ToString() As String
```

C#

```
public override string  ToString();
```

Return value

A string representing the InfoMessage event.

See also

- "SAInfoMessageEventArgs class" on page 344
- "SAInfoMessageEventArgs members" on page 344

SAInfoMessageEventHandler delegate

Represents the method that handles the SAConnection.InfoMessage event of an SAConnection object.

Syntax

Visual Basic

```
Public Delegate Sub SAInfoMessageEventHandler( _
  ByVal obj As Object, _
  ByVal args As SAInfoMessageEventArgs _
)
```

C#

```
public delegate void SAInfoMessageEventHandler(
  object obj,
  SAInfoMessageEventArgs args
);
```

See also

- "SAConnection class" on page 231
- "InfoMessage event" on page 247

SAIsolationLevel enumeration

Specifies SQL Anywhere isolation levels. This class augments the IsolationLevel.

Syntax

Visual Basic

Public Enum **SAIsolationLevel**

C#

public enum **SAIsolationLevel**

Remarks

The SQL Anywhere .NET Data Provider supports all SQL Anywhere isolation levels, including the snapshot isolation levels. To use snapshot isolation, specify one of SAIsolationLevel.Snapshot, SAIsolationLevel.ReadOnlySnapshot, or SAIsolationLevel.StatementSnapshot as the parameter to BeginTransaction. BeginTransaction has been overloaded so it can take either an IsolationLevel or an SAIsolationLevel. The values in the two enumerations are the same, except for ReadOnlySnapshot and StatementS napshot which exist only in SAIsolationLevel. There is a new property in SATransaction called SAIsolationLevel that gets the SAIsolationLevel.

For more information, see "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Members

Member name	Description	Value
Chaos	This isolation level is unsupported.	16
ReadCommitted	Sets the behavior to be equivalent to isolation level 1.	4096
ReadOnlySnapshot	For read-only statements, use a snapshot of committed data from the time when the first row is read from the database.	16777217
ReadUncommitted	Sets the behavior to be equivalent to isolation level 0.	256
RepeatableRead	Sets the behavior to be equivalent to isolation level 2.	65536
Serializable	Sets the behavior to be equivalent to isolation level 3.	1048576
Snapshot	Uses a snapshot of committed data from the time when the first row is read, inserted, updated, or deleted by the transaction.	16777216

Member name	Description	Value
StatementSnapshot	Use a snapshot of committed data from the time when the first row is read by the statement. Each statement within the transaction sees a snapshot of data from a different time.	16777218
Unspecified	This isolation level is unsupported.	-1

SAMessageType enumeration

Identifies the type of message. This can be one of: Action, Info, Status, or Warning.

Syntax

Visual Basic

Public Enum **SAMessageType**

C#

public enum **SAMessageType**

Members

Member name	Description	Value
Action	Message of type ACTION.	2
Info	Message of type INFO.	0
Status	Message of type STATUS.	3
Warning	Message of type WARNING.	1

SAMetaDataCollectionNames class

Provides a list of constants for use with the SAConnection.GetSchema(String,String[]) method to retrieve metadata collections. This class cannot be inherited.

Syntax

Visual Basic

Public NotInheritable Class **SAMetaDataCollectionNames**

C#

public sealed class **SAMetaDataCollectionNames**

Remarks

This field is constant and read-only.

See also

- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

SAMetaDataCollectionNames members

Public fields

Member name	Description
Columns field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Columns collection. This field is read-only.
DataSourceInformation field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the DataSourceInformation collection. This field is read-only.
DataTypes field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the DataTypes collection. This field is read-only.
ForeignKeys field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the ForeignKeys collection. This field is read-only.
IndexColumns field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the IndexColumns collection. This field is read-only.
Indexes field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Indexes collection. This field is read-only.
MetaDataCollections field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the MetaDataCollections collection. This field is read-only.
ProcedureParameters field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the ProcedureParameters collection. This field is read-only.

Member name	Description
Procedures field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Procedures collection. This field is read-only.
ReservedWords field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the ReservedWords collection. This field is read-only.
Restrictions field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Restrictions collection. This field is read-only.
Tables field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Tables collection. This field is read-only.
UserDefinedTypes field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the UserDefinedTypes collection. This field is read-only.
Users field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Users collection. This field is read-only.
ViewColumns field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the ViewColumns collection. This field is read-only.
Views field	Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Views collection. This field is read-only.

See also

- "SAMetaDataCollectionNames class" on page 349
- "GetSchema(String, String[]) method" on page 246

Columns field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Columns collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **Columns** As String

C#

public const string **Columns** ;

Example

The following code fills a DataTable with the Columns collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Columns );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

DataSourceInformation field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the DataSourceInformation collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **DataSourceInformation** As String

C#

public const string **DataSourceInformation** ;

Example

The following code fills a DataTable with the DataSourceInformation collection.

```
DataTable schema =
GetSchema( SAMetaDataCollectionNames.DataSourceInformation );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

DataTypes field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the DataTypes collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **DataTypes** As String

C#

public const string **DataTypes** ;

Example

The following code fills a DataTable with the DataTypes collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.DataTypes );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

ForeignKeys field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the ForeignKeys collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **ForeignKeys** As String

C#

public const string **ForeignKeys** ;

Example

The following code fills a DataTable with the ForeignKeys collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.ForeignKeys );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

IndexColumns field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the IndexColumns collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **IndexColumns** As String

C#

public const string **IndexColumns** ;

Example

The following code fills a DataTable with the IndexColumns collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.IndexColumns );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

Indexes field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Indexes collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **Indexes** As String

C#

public const string **Indexes** ;

Example

The following code fills a DataTable with the Indexes collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Indexes );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

MetaDataCollections field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the MetaDataCollections collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **MetaDataCollections** As String

C#

public const string **MetaDataCollections** ;

Example

The following code fills a DataTable with the MetaDataCollections collection.

```
DataTable schema =
GetSchema( SAMetaDataCollectionNames.MetaDataCollections );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

ProcedureParameters field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the ProcedureParameters collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **ProcedureParameters** As String

C#

public const string **ProcedureParameters** ;

Example

The following code fills a DataTable with the ProcedureParameters collection.

```
DataTable schema =
GetSchema( SAMetaDataCollectionNames.ProcedureParameters );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

Procedures field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Procedures collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **Procedures** As String

C#

public const string **Procedures** ;

Example

The following code fills a DataTable with the Procedures collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Procedures );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

ReservedWords field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the ReservedWords collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **ReservedWords** As String

C#

public const string **ReservedWords** ;

Example

The following code fills a DataTable with the ReservedWords collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.ReservedWords );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

Restrictions field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Restrictions collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **Restrictions** As String

C#

public const string **Restrictions** ;

Example

The following code fills a DataTable with the Restrictions collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Restrictions );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

Tables field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Tables collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **Tables** As String

C#

public const string **Tables** ;

Example

The following code fills a DataTable with the Tables collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Tables );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

UserDefinedTypes field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the UserDefinedTypes collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **UserDefinedTypes** As String

C#

public const string **UserDefinedTypes** ;

Example

The following code fills a DataTable with the Users collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.UserDefinedTypes );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

Users field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Users collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **Users** As String

C#

public const string **Users** ;

Example

The following code fills a DataTable with the Users collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Users );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

ViewColumns field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the ViewColumns collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **ViewColumns** As String

C#

public const string **ViewColumns** ;

Example

The following code fills a DataTable with the ViewColumns collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.ViewColumns );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

Views field

Provides a constant for use with the SAConnection.GetSchema(String,String[]) method that represents the Views collection. This field is read-only.

Syntax

Visual Basic

Public Shared Readonly **Views** As String

C#

public const string **Views** ;

Example

The following code fills a DataTable with the Views collection.

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Views );
```

See also

- "SAMetaDataCollectionNames class" on page 349
- "SAMetaDataCollectionNames members" on page 350
- "GetSchema(String, String[]) method" on page 246

SAParameter class

Represents a parameter to an SACommand, and optionally, its mapping to a DataSet column. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SAParameter
  Inherits DbParameter
  Implements ICloneable
```

C#

```
public sealed class SAParameter : DbParameter,
  ICloneable
```

Remarks

Implements:IDbDataParameter, IDataParameter, ICloneable

See also

- "SAParameter members" on page 360

SAParameter members

Public constructors

Member name	Description
SAParameter constructors	Initializes a new instance of the "SAParameter class" on page 359.

Public properties

Member name	Description
DbType property	Gets and sets the DbType of the parameter.
Direction property	Gets and sets a value indicating whether the parameter is input-only, output-only, bidirectional, or a stored procedure return value parameter.
IsNullable property	Gets and sets a value indicating whether the parameter accepts null values.
Offset property	Gets and sets the offset to the Value property.
ParameterName property	Gets and sets the name of the SAParameter.
Precision property	Gets and sets the maximum number of digits used to represent the Value property.
SADbType property	The SADbType of the parameter.

Member name	Description
Scale property	Gets and sets the number of decimal places to which Value is resolved.
Size property	Gets and sets the maximum size, in bytes, of the data within the column.
SourceColumn property	Gets and sets the name of the source column mapped to the DataSet and used for loading or returning the value.
SourceColumnNullMapping property	Gets and sets value that indicates whether the source column is nullable. This allows SACommandBuilder to generate Update statements for nullable columns correctly.
SourceVersion property	Gets and sets the DataRowVersion to use when loading Value.
Value property	Gets and sets the value of the parameter.

Public methods

Member name	Description
ResetDbType method	Resets the type (the values of DbType and SADbType) associated with this SAParameter.
ToString method	Returns a string containing the ParameterName.

See also

- "SAParameter class" on page 359

SAParameter constructors

Initializes a new instance of the "SAParameter class" on page 359.

SAParameter() constructor

Initializes an SAParameter object with null (Nothing in Visual Basic) as its value.

Syntax

Visual Basic

Public Sub **New()**

C#

public **SAParameter();**

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360
- "SAParameter constructors" on page 361

SAParameter(String, Object) constructor

Initializes an SAParameter object with the specified parameter name and value. This constructor is not recommended; it is provided for compatibility with other data providers.

Syntax

Visual Basic

```
Public Sub New( _
   ByVal parameterName As String, _
   ByVal value As Object _
)
```

C#

```
public  SAParameter(
   string  parameterName,
   object value
);
```

Parameters

- **parameterName** The name of the parameter.
- **value** An Object that is the value of the parameter.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360
- "SAParameter constructors" on page 361

SAParameter(String, SADbType) constructor

Initializes an SAParameter object with the specified parameter name and data type.

Syntax

Visual Basic

```
Public Sub New( _
   ByVal parameterName As String, _
   ByVal dbType As SADbType _
)
```

C#


```
public  SAParameter(
  string  parameterName,
  SADbType dbType
);
```

Parameters

- **parameterName** The name of the parameter.
- **dbType** One of the SADbType values.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360
- "SAParameter constructors" on page 361
- "SADbType property" on page 369

SAParameter(String, SADbType, Int32) constructor

Initializes an SAParameter object with the specified parameter name and data type.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal dbType As SADbType, _
  ByVal size As Integer _
)
```

C#

```
public  SAParameter(
  string  parameterName,
  SADbType dbType,
  int size
);
```

Parameters

- **parameterName** The name of the parameter.
- **dbType** One of the SADbType values
- **size** The length of the parameter.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360
- "SAParameter constructors" on page 361

SAParameter(String, SADbType, Int32, String) constructor

Initializes an SAParameter object with the specified parameter name, data type, and length.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal dbType As SADbType, _
  ByVal size As Integer, _
  ByVal sourceColumn As String _
)
```

C#

```
public  SAParameter(
  string  parameterName,
  SADbType dbType,
  int size,
  string  sourceColumn
);
```

Parameters

- **parameterName** The name of the parameter.
- **dbType** One of the SADbType values
- **size** The length of the parameter.
- **sourceColumn** The name of the source column to map.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360
- "SAParameter constructors" on page 361

SAParameter(String, SADbType, Int32, ParameterDirection, Boolean, Byte, Byte, String, DataRowVersion, Object) constructor

Initializes an SAParameter object with the specified parameter name, data type, length, direction, nullability, numeric precision, numeric scale, source column, source version, and value.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal dbType As SADbType, _
  ByVal size As Integer, _
  ByVal direction As ParameterDirection, _
  ByVal isNullable As Boolean, _
```



```
    ByVal precision As Byte, _
    ByVal scale As Byte, _
    ByVal sourceColumn As String, _
    ByVal sourceVersion As DataRowVersion, _
    ByVal value As Object _
)

C#

public SAParameter(
    string parameterName,
    SADbType dbType,
    int size,
    ParameterDirection direction,
    bool isNullable,
    byte precision,
    byte scale,
    string sourceColumn,
    DataRowVersion sourceVersion,
    object value
);
```

Parameters

- **parameterName** The name of the parameter.
- **dbType** One of the SADbType values
- **size** The length of the parameter.
- **direction** One of the ParameterDirection values.
- **isNullable** True if the value of the field can be null; otherwise, false.
- **precision** The total number of digits to the left and right of the decimal point to which Value is resolved.
- **scale** The total number of decimal places to which Value is resolved.
- **sourceColumn** The name of the source column to map.
- **sourceVersion** One of the DataRowVersion values.
- **value** An Object that is the value of the parameter.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360
- "SAParameter constructors" on page 361

DbType property

Gets and sets the DbType of the parameter.

Syntax

Visual Basic

Public Overrides Property **DbType** As DbType

C#

public override DbType **DbType** { get; set; }

Remarks

The SADbType and DbType are linked. Therefore, setting the DbType changes the SADbType to a supporting SADbType.

The value must be a member of the SADbType enumerator.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

Direction property

Gets and sets a value indicating whether the parameter is input-only, output-only, bidirectional, or a stored procedure return value parameter.

Syntax

Visual Basic

Public Overrides Property **Direction** As ParameterDirection

C#

public override ParameterDirection **Direction** { get; set; }

Property value

One of the ParameterDirection values.

Remarks

If the ParameterDirection is output, and execution of the associated SACommand does not return a value, the SAParameter contains a null value. After the last row from the last result set is read, the Output, InputOut, and ReturnValue parameters are updated.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

IsNullable property

Gets and sets a value indicating whether the parameter accepts null values.

Syntax

Visual Basic

Public Overrides Property **IsNullable** As Boolean

C#

public override bool **IsNullable** { get; set; }

Remarks

This property is true if null values are accepted; otherwise, it is false. The default is false. Null values are handled using the DBNull class.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

Offset property

Gets and sets the offset to the Value property.

Syntax

Visual Basic

Public Property **Offset** As Integer

C#

public int **Offset** { get; set; }

Property value

The offset to the value. The default is 0.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

ParameterName property

Gets and sets the name of the SAParameter.

Syntax

Visual Basic

Public Overrides Property **ParameterName** As String

C#

public override string **ParameterName** { get; set; }

Property value

The default is an empty string.

Remarks

The SQL Anywhere .NET Data Provider uses positional parameters that are marked with a question mark (?) instead of named parameters.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

Precision property

Gets and sets the maximum number of digits used to represent the Value property.

Syntax

Visual Basic

Public Property **Precision** As Byte

C#

public byte **Precision** { get; set; }

Property value

The value of this property is the maximum number of digits used to represent the Value property. The default value is 0, which indicates that the data provider sets the precision for the Value property.

Remarks

The Precision property is only used for decimal and numeric input parameters.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

SADbType property

The SADbType of the parameter.

Syntax

Visual Basic

Public Property **SADbType** As SADbType

C#

public SADbType **SADbType** { get; set; }

Remarks

The SADbType and DbType are linked. Therefore, setting the SADbType changes the DbType to a supporting DbType.

The value must be a member of the SADbType enumerator.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

Scale property

Gets and sets the number of decimal places to which Value is resolved.

Syntax

Visual Basic

Public Property **Scale** As Byte

C#

public byte **Scale** { get; set; }

Property value

The number of decimal places to which Value is resolved. The default is 0.

Remarks

The Scale property is only used for decimal and numeric input parameters.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

Size property

Gets and sets the maximum size, in bytes, of the data within the column.

Syntax

Visual Basic

Public Overrides Property **Size** As Integer

C#

public override int **Size** { get; set; }

Property value

The value of this property is the maximum size, in bytes, of the data within the column. The default value is inferred from the parameter value.

Remarks

The value of this property is the maximum size, in bytes, of the data within the column. The default value is inferred from the parameter value.

The Size property is used for binary and string types.

For variable length data types, the Size property describes the maximum amount of data to transmit to the server. For example, the Size property can be used to limit the amount of data sent to the server for a string value to the first one hundred bytes.

If not explicitly set, the size is inferred from the actual size of the specified parameter value. For fixed width data types, the value of Size is ignored. It can be retrieved for informational purposes, and returns the maximum amount of bytes the provider uses when transmitting the value of the parameter to the server.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

SourceColumn property

Gets and sets the name of the source column mapped to the DataSet and used for loading or returning the value.

Syntax

Visual Basic

Public Overrides Property **SourceColumn** As String

C#

public override string **SourceColumn** { get; set; }

Property value

A string specifying the name of the source column mapped to the DataSet and used for loading or returning the value.

Remarks

When SourceColumn is set to anything other than an empty string, the value of the parameter is retrieved from the column with the SourceColumn name. If Direction is set to Input, the value is taken from the DataSet. If Direction is set to Output, the value is taken from the data source. A Direction of InputOutput is a combination of both.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

SourceColumnNullMapping property

Gets and sets value that indicates whether the source column is nullable. This allows SACommandBuilder to generate Update statements for nullable columns correctly.

Syntax

Visual Basic

Public Overrides Property **SourceColumnNullMapping** As Boolean

C#

public override bool **SourceColumnNullMapping** { get; set; }

Remarks

If the source column is nullable, true is returned; otherwise, false.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

SourceVersion property

Gets and sets the DataRowVersion to use when loading Value.

Syntax

Visual Basic

Public Overrides Property **SourceVersion** As DataRowVersion

C#

public override DataRowVersion **SourceVersion** { get; set; }

Remarks

Used by UpdateCommand during an Update operation to determine whether the parameter value is set to Current or Original. This allows primary keys to be updated. This property is ignored by InsertCommand and DeleteCommand. This property is set to the version of the DataRow used by the Item property, or the GetChildRows method of the DataRow object.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

Value property

Gets and sets the value of the parameter.

Syntax

Visual Basic

Public Overrides Property **Value** As Object

C#

public override object **Value** { get; set; }

Property value

An Object that specifies the value of the parameter.

Remarks

For input parameters, the value is bound to the SACommand that is sent to the server. For output and return value parameters, the value is set on completion of the SACommand and after the SADataReader is closed.

When sending a null parameter value to the server, you must specify DBNull, not null. The null value in the system is an empty object that has no value. DBNull is used to represent null values.

If the application specifies the database type, the bound value is converted to that type when the SQL Anywhere .NET Data Provider sends the data to the server. The provider attempts to convert any type of value if it supports the IConvertible interface. Conversion errors may result if the specified type is not compatible with the value.

Both the DbType and SADbType properties can be inferred by setting the Value.

The Value property is overwritten by Update.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

ResetDbType method

Resets the type (the values of DbType and SADbType) associated with this SAParameter.

Syntax

Visual Basic

Public Overrides Sub **ResetDbType()**

C#

public override void **ResetDbType();**

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

ToString method

Returns a string containing the ParameterName.

Syntax

Visual Basic

Public Overrides Function **ToString()** As String

C#

public override string **ToString();**

Return value

The name of the parameter.

See also

- "SAParameter class" on page 359
- "SAParameter members" on page 360

SAParameterCollection class

Represents all parameters to an SACommand object and, optionally, their mapping to a DataSet column. This class cannot be inherited.

Syntax

Visual Basic

Public NotInheritable Class **SAParameterCollection**
  Inherits DbParameterCollection

C#

public sealed class **SAParameterCollection** : DbParameterCollection

Remarks

There is no constructor for SAParameterCollection. You obtain an SAParameterCollection object from the SACommand.Parameters property of an SACommand oject.

See also

- "SAParameterCollection members" on page 374
- "SACommand class" on page 192
- "Parameters property" on page 199
- "SAParameter class" on page 359
- "SAParameterCollection class" on page 373

SAParameterCollection members

Public properties

Member name	Description
Count property	Returns the number of SAParameter objects in the collection.
IsFixedSize property	Gets a value that indicates whether the SAParameterCollection has a fixed size.
IsReadOnly property	Gets a value that indicates whether the SAParameterCollection is read-only.
IsSynchronized property	Gets a value that indicates whether the SAParameterCollection object is synchronized.
Item properties	Gets and sets the SAParameter object at the specified index.
SyncRoot property	Gets an object that can be used to synchronize access to the SAParameterCollection.

Public methods

Member name	Description
Add methods	Adds an SAParameter object to this collection.
AddRange methods	Adds an array of values to the end of the SAParameterCollection.
AddWithValue method	Adds a value to the end of this collection.
Clear method	Removes all items from the collection.

Member name	Description
Contains methods	Indicates whether an SAParameter object exists in the collection.
CopyTo method	Copies SAParameter objects from the SAParameterCollection to the specified array.
GetEnumerator method	Returns an enumerator that iterates through the SAParameterCollection.
IndexOf methods	Returns the location of the SAParameter object in the collection.
Insert method	Inserts an SAParameter object in the collection at the specified index.
Remove method	Removes the specified SAParameter object from the collection.
RemoveAt methods	Removes the specified SAParameter object from the collection.

See also

- "SAParameterCollection class" on page 373
- "SACommand class" on page 192
- "Parameters property" on page 199
- "SAParameter class" on page 359
- "SAParameterCollection class" on page 373

Count property

Returns the number of SAParameter objects in the collection.

Syntax

Visual Basic

```
Public Overrides Readonly Property Count As Integer
```

C#

```
public override int Count  { get;}
```

Property value

The number of SAParameter objects in the collection.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "SAParameter class" on page 359
- "SAParameterCollection class" on page 373

IsFixedSize property

Gets a value that indicates whether the SAParameterCollection has a fixed size.

Syntax

Visual Basic

Public Overrides Readonly Property **IsFixedSize** As Boolean

C#

public override bool **IsFixedSize** { get;}

Property value

True if this collection has a fixed size, false otherwise.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374

IsReadOnly property

Gets a value that indicates whether the SAParameterCollection is read-only.

Syntax

Visual Basic

Public Overrides Readonly Property **IsReadOnly** As Boolean

C#

public override bool **IsReadOnly** { get;}

Property value

True if this collection is read-only, false otherwise.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374

IsSynchronized property

Gets a value that indicates whether the SAParameterCollection object is synchronized.

Syntax

Visual Basic

```
Public Overrides Readonly Property IsSynchronized As Boolean
```

C#

```
public override bool IsSynchronized  { get;}
```

Property value

True if this collection is synchronized, false otherwise.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374

Item properties

Gets and sets the SAParameter object at the specified index.

Item(Int32) property

Gets and sets the SAParameter object at the specified index.

Syntax

Visual Basic

```
Public Property Item ( _
   ByVal index As Integer _
) As SAParameter
```

C#

```
public SAParameter this [
   int index
] { get; set; }
```

Parameters

- **index** The zero-based index of the parameter to retrieve.

Property value

The SAParameter at the specified index.

Remarks

In C#, this property is the indexer for the SAParameterCollection object.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "Item properties" on page 377
- "SAParameter class" on page 359
- "SAParameterCollection class" on page 373

Item(String) property

Gets and sets the SAParameter object at the specified index.

Syntax

Visual Basic

```
Public Property Item ( _
   ByVal parameterName As String _
) As SAParameter
```

C#

```
public SAParameter this [
   string  parameterName
] { get; set; }
```

Parameters

- **parameterName** The name of the parameter to retrieve.

Property value

The SAParameter object with the specified name.

Remarks

In C#, this property is the indexer for the SAParameterCollection object.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "Item properties" on page 377
- "SAParameter class" on page 359
- "SAParameterCollection class" on page 373
- "Item(Int32) property" on page 294
- "GetOrdinal method" on page 309
- "GetValue(Int32) method" on page 315
- "GetFieldType method" on page 304

SyncRoot property

Gets an object that can be used to synchronize access to the SAParameterCollection.

Syntax

Visual Basic

```
Public Overrides Readonly Property SyncRoot As Object
```

C#

```
public override object SyncRoot  { get;}
```

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374

Add methods

Adds an SAParameter object to this collection.

Add(Object) method

Adds an SAParameter object to this collection.

Syntax

Visual Basic

```
Public Overrides Function Add( _
   ByVal value As Object _
) As Integer
```

C#

```
public override int Add(
   object value
);
```

Parameters

- **value** The SAParameter object to add to the collection.

Return value

The index of the new SAParameter object.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374

- "Add methods" on page 379
- "SAParameter class" on page 359

Add(SAParameter) method

Adds an SAParameter object to this collection.

Syntax

Visual Basic

```
Public Function Add( _
  ByVal value As SAParameter _
) As SAParameter
```

C#

```
public SAParameter Add(
  SAParameter value
);
```

Parameters

- **value** The SAParameter object to add to the collection.

Return value

The new SAParameter object.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "Add methods" on page 379

Add(String, Object) method

Adds an SAParameter object to this collection, created using the specified parameter name and value, to the collection.

Syntax

Visual Basic

```
Public Function Add( _
  ByVal parameterName As String, _
  ByVal value As Object _
) As SAParameter
```

C#

```
public SAParameter Add(
  string  parameterName,
```



```
  object value
);
```

Parameters

- **parameterName** The name of the parameter.
- **value** The value of the parameter to add to the connection.

Return value

The new SAParameter object.

Remarks

Because of the special treatment of the 0 and 0.0 constants and the way overloaded methods are resolved, it is highly recommended that you explicitly cast constant values to type object when using this method.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "Add methods" on page 379
- "SAParameter class" on page 359

Add(String, SADbType) method

Adds an SAParameter object to this collection, created using the specified parameter name and data type, to the collection.

Syntax

Visual Basic

```
Public Function Add( _
  ByVal parameterName As String, _
  ByVal saDbType As SADbType _
) As SAParameter
```

C#

```
public SAParameter Add(
  string  parameterName,
  SADbType saDbType
);
```

Parameters

- **parameterName** The name of the parameter.
- **saDbType** One of the SADbType values.

Return value

The new SAParameter object.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "Add methods" on page 379
- "SADbType enumeration" on page 322
- "Add(SAParameter) method" on page 380
- "Add(String, Object) method" on page 380

Add(String, SADbType, Int32) method

Adds an SAParameter object to this collection, created using the specified parameter name, data type, and length, to the collection.

Syntax

Visual Basic

```
Public Function Add( _
   ByVal parameterName As String, _
   ByVal saDbType As SADbType, _
   ByVal size As Integer _
) As SAParameter
```

C#

```
public SAParameter Add(
   string  parameterName,
   SADbType saDbType,
   int size
);
```

Parameters

- **parameterName** The name of the parameter.
- **saDbType** One of the SADbType values.
- **size** The length of the parameter.

Return value

The new SAParameter object.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "Add methods" on page 379
- "SADbType enumeration" on page 322
- "Add(SAParameter) method" on page 380
- "Add(String, Object) method" on page 380

Add(String, SADbType, Int32, String) method

Adds an SAParameter object to this collection, created using the specified parameter name, data type, length, and source column name, to the collection.

Syntax

Visual Basic

```
Public Function Add( _
   ByVal parameterName As String, _
   ByVal saDbType As SADbType, _
   ByVal size As Integer, _
   ByVal sourceColumn As String _
) As SAParameter
```

C#

```
public SAParameter Add(
   string  parameterName,
   SADbType saDbType,
   int size,
   string  sourceColumn
);
```

Parameters

- **parameterName** The name of the parameter.
- **saDbType** One of the SADbType values.
- **size** The length of the column.
- **sourceColumn** The name of the source column to map.

Return value

The new SAParameter object.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "Add methods" on page 379
- "SADbType enumeration" on page 322
- "Add(SAParameter) method" on page 380
- "Add(String, Object) method" on page 380

AddRange methods

Adds an array of values to the end of the SAParameterCollection.

AddRange(Array) method

Adds an array of values to the end of the SAParameterCollection.

Syntax

Visual Basic

```
Public Overrides Sub AddRange( _
  ByVal values As Array _
)
```

C#

```
public override void AddRange(
  Array values
);
```

Parameters

- **values** The values to add.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "AddRange methods" on page 383

AddRange(SAParameter[]) method

Adds an array of values to the end of the SAParameterCollection.

Syntax

Visual Basic

```
Public Sub AddRange( _
  ByVal values As SAParameter() _
)
```

C#

```
public void AddRange(
  SAParameter[] values
);
```

Parameters

- **values** An array of SAParameter objects to add to the end of this collection.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "AddRange methods" on page 383

AddWithValue method

Adds a value to the end of this collection.

Syntax

Visual Basic

```
Public Function AddWithValue( _
  ByVal parameterName As String, _
  ByVal value As Object _
) As SAParameter
```

C#

```
public SAParameter AddWithValue(
  string  parameterName,
  object value
);
```

Parameters

- **parameterName** The name of the parameter.
- **value** The value to be added.

Return value

The new SAParameter object.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374

Clear method

Removes all items from the collection.

Syntax

Visual Basic

```
Public Overrides Sub Clear()
```

C#

```
public override void Clear();
```

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374

Contains methods

Indicates whether an SAParameter object exists in the collection.

Contains(Object) method

Indicates whether an SAParameter object exists in the collection.

Syntax

Visual Basic

```
Public Overrides Function Contains( _
   ByVal value As Object _
) As Boolean
```

C#

```
public override bool Contains(
   object value
);
```

Parameters

- **value** The SAParameter object to find.

Return value

True if the collection contains the SAParameter object. Otherwise, false.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "Contains methods" on page 386
- "SAParameter class" on page 359
- "Contains(String) method" on page 386

Contains(String) method

Indicates whether an SAParameter object exists in the collection.

Syntax

Visual Basic

```
Public Overrides Function Contains( _
   ByVal value As String _
) As Boolean
```

C#

```
public override bool Contains(
```



```
  string  value
);
```

Parameters

- **value** The name of the parameter to search for.

Return value

True if the collection contains the SAParameter object. Otherwise, false.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "Contains methods" on page 386
- "SAParameter class" on page 359
- "Contains(Object) method" on page 386

CopyTo method

Copies SAParameter objects from the SAParameterCollection to the specified array.

Syntax

Visual Basic

```
Public Overrides Sub CopyTo( _
  ByVal array As Array, _
  ByVal index As Integer _
)
```

C#

```
public override void CopyTo(
  Array array,
  int index
);
```

Parameters

- **array** The array to copy the SAParameter objects into.
- **index** The starting index of the array.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "SAParameter class" on page 359
- "SAParameterCollection class" on page 373

GetEnumerator method

Returns an enumerator that iterates through the SAParameterCollection.

Syntax

Visual Basic

```
Public Overrides Function GetEnumerator() As IEnumerator
```

C#

```
public override IEnumerator GetEnumerator();
```

Return value

An IEnumerator for the SAParameterCollection object.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "SAParameterCollection class" on page 373

IndexOf methods

Returns the location of the SAParameter object in the collection.

IndexOf(Object) method

Returns the location of the SAParameter object in the collection.

Syntax

Visual Basic

```
Public Overrides Function IndexOf( _
   ByVal value As Object _
) As Integer
```

C#

```
public override int IndexOf(
   object value
);
```

Parameters

- **value** The SAParameter object to locate.

Return value

The zero-based location of the SAParameter object in the collection.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "IndexOf methods" on page 388
- "SAParameter class" on page 359
- "IndexOf(String) method" on page 389

IndexOf(String) method

Returns the location of the SAParameter object in the collection.

Syntax

Visual Basic

```
Public Overrides Function IndexOf( _
  ByVal parameterName As String _
) As Integer
```

C#

```
public override int IndexOf(
  string  parameterName
);
```

Parameters

- **parameterName** The name of the parameter to locate.

Return value

The zero-based index of the SAParameter object in the collection.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "IndexOf methods" on page 388
- "SAParameter class" on page 359
- "IndexOf(Object) method" on page 388

Insert method

Inserts an SAParameter object in the collection at the specified index.

Syntax

Visual Basic

```
Public Overrides Sub Insert( _
  ByVal index As Integer, _
```

```
  ByVal value As Object _
)
```

C#

```
public override void Insert(
  int index,
  object value
);
```

Parameters

- **index** The zero-based index where the parameter is to be inserted within the collection.
- **value** The SAParameter object to add to the collection.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374

Remove method

Removes the specified SAParameter object from the collection.

Syntax

Visual Basic

```
Public Overrides Sub Remove( _
  ByVal value As Object _
)
```

C#

```
public override void Remove(
  object value
);
```

Parameters

- **value** The SAParameter object to remove from the collection.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374

RemoveAt methods

Removes the specified SAParameter object from the collection.

RemoveAt(Int32) method

Removes the specified SAParameter object from the collection.

Syntax

Visual Basic

```
Public Overrides Sub RemoveAt( _
  ByVal index As Integer _
)
```

C#

```
public override void RemoveAt(
  int index
);
```

Parameters

- **index** The zero-based index of the parameter to remove.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374
- "RemoveAt methods" on page 390
- "RemoveAt(String) method" on page 391

RemoveAt(String) method

Removes the specified SAParameter object from the collection.

Syntax

Visual Basic

```
Public Overrides Sub RemoveAt( _
  ByVal parameterName As String _
)
```

C#

```
public override void RemoveAt(
  string  parameterName
);
```

Parameters

- **parameterName** The name of the SAParameter object to remove.

See also

- "SAParameterCollection class" on page 373
- "SAParameterCollection members" on page 374

- "RemoveAt methods" on page 390
- "RemoveAt(Int32) method" on page 391

SAPermission class

Enables the SQL Anywhere .NET Data Provider to ensure that a user has a security level adequate to access a SQL Anywhere data source. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SAPermission
  Inherits DBDataPermission
```

C#

```
public sealed class SAPermission : DBDataPermission
```

Remarks

Base classes DBDataPermission

See also

- "SAPermission members" on page 392

SAPermission members

Public constructors

Member name	Description
SAPermission constructor	Initializes a new instance of the SAPermission class.

Public properties

Member name	Description
AllowBlankPassword (inherited from DBDataPermission)	Gets a value indicating whether a blank password is allowed.

Public methods

Member name	Description
Add (inherited from DBDataPermission)	Adds access for the specified connection string to the existing state of the DBDataPermission.

Member name	Description
Assert (inherited from CodeAccessPermission)	
Copy (inherited from DBDataPermission)	Creates and returns an identical copy of the current permission object.
Demand (inherited from CodeAccessPermission)	
Deny (inherited from CodeAccessPermission)	
Equals (inherited from CodeAccessPermission)	
FromXml (inherited from DBDataPermission)	Reconstructs a security object with a specified state from an XML encoding.
GetHashCode (inherited from CodeAccessPermission)	
Intersect (inherited from DBDataPermission)	Returns a new permission object representing the intersection of the current permission object and the specified permission object.
IsSubsetOf (inherited from DBDataPermission)	Returns a value indicating whether the current permission object is a subset of the specified permission object.
IsUnrestricted (inherited from DBDataPermission)	Returns a value indicating whether the permission can be represented as unrestricted without any knowledge of the permission semantics.
PermitOnly (inherited from CodeAccessPermission)	
ToString (inherited from CodeAccessPermission)	
ToXml (inherited from DBDataPermission)	Creates an XML encoding of the security object and its current state.
Union (inherited from DBDataPermission)	Returns a new permission object that is the union of the current and specified permission objects.

See also

- "SAPermission class" on page 392

SAPermission constructor

Initializes a new instance of the SAPermission class.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal state As PermissionState _
)
```

C#

```
public  SAPermission(
  PermissionState state
);
```

Parameters

- **state** One of the PermissionState values.

See also

- "SAPermission class" on page 392
- "SAPermission members" on page 392

SAPermissionAttribute class

Associates a security action with a custom security attribute. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SAPermissionAttribute
  Inherits DBDataPermissionAttribute
```

C#

```
public sealed class SAPermissionAttribute : DBDataPermissionAttribute
```

See also

- "SAPermissionAttribute members" on page 395

SAPermissionAttribute members

Public constructors

Member name	Description
SAPermissionAttribute constructor	Initializes a new instance of the SAPermissionAttribute class.

Public properties

Member name	Description
Action (inherited from SecurityAttribute)	
AllowBlankPassword (inherited from DBDataPermissionAttribute)	Gets or sets a value indicating whether a blank password is allowed.
ConnectionString (inherited from DBDataPermissionAttribute)	Gets or sets a permitted connection string.
KeyRestrictionBehavior (inherited from DBDataPermissionAttribute)	Identifies whether the list of connection string parameters identified by the DBDataPermissionAttribute.KeyRestrictions are the only connection string parameters allowed.
KeyRestrictions (inherited from DBDataPermissionAttribute)	Gets or sets connection string parameters that are allowed or disallowed.
TypeId (inherited from Attribute)	
Unrestricted (inherited from SecurityAttribute)	

Public methods

Member name	Description
CreatePermission method	Returns an SAPermission object that is configured according to the attribute properties.
Equals (inherited from Attribute)	
GetHashCode (inherited from Attribute)	

Member name	Description
IsDefaultAttribute (inherited from Attribute)	
Match (inherited from Attribute)	
ShouldSerializeConnection-String (inherited from DBData-PermissionAttribute)	Identifies whether the attribute should serialize the connection string.
ShouldSerializeKeyRestrictions (inherited from DBDataPermis-sionAttribute)	Identifies whether the attribute should serialize the set of key restric-tions.

See also

- "SAPermissionAttribute class" on page 394

SAPermissionAttribute constructor

Initializes a new instance of the SAPermissionAttribute class.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal action As SecurityAction _
)
```

C#

```
public  SAPermissionAttribute(
  SecurityAction action
);
```

Parameters

- **action** One of the SecurityAction values representing an action that can be performed using declarative security.

See also

- "SAPermissionAttribute class" on page 394
- "SAPermissionAttribute members" on page 395

CreatePermission method

Returns an SAPermission object that is configured according to the attribute properties.

Syntax

Visual Basic

Public Overrides Function **CreatePermission()** As IPermission

C#

public override IPermission **CreatePermission();**

See also

- "SAPermissionAttribute class" on page 394
- "SAPermissionAttribute members" on page 395

SARowsCopiedEventArgs class

Represents the set of arguments passed to the SARowsCopiedEventHandler. This class cannot be inherited.

Syntax

Visual Basic

Public NotInheritable Class **SARowsCopiedEventArgs**

C#

public sealed class **SARowsCopiedEventArgs**

Remarks

Restrictions: The SARowsCopiedEventArgs class is not available in the .NET Compact Framework 2.0.

See also

- "SARowsCopiedEventArgs members" on page 397

SARowsCopiedEventArgs members

Public constructors

Member name	Description
SARowsCopiedEventArgs constructor	Creates a new instance of the SARowsCopiedEventArgs object.

Public properties

Member name	Description
Abort property	Gets or sets a value that indicates whether the bulk-copy operation should be aborted.
RowsCopied property	Gets the number of rows copied during the current bulk-copy operation.

See also

- "SARowsCopiedEventArgs class" on page 397

SARowsCopiedEventArgs constructor

Creates a new instance of the SARowsCopiedEventArgs object.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal rowsCopied As Long _
)
```

C#

```
public  SARowsCopiedEventArgs(
  long rowsCopied
);
```

Parameters

- **rowsCopied** An 64-bit integer value that indicates the number of rows copied during the current bulk-copy operation.

Remarks

Restrictions: The SARowsCopiedEventArgs class is not available in the .NET Compact Framework 2.0.

See also

- "SARowsCopiedEventArgs class" on page 397
- "SARowsCopiedEventArgs members" on page 397

Abort property

Gets or sets a value that indicates whether the bulk-copy operation should be aborted.

Syntax

Visual Basic

```
Public Property Abort As Boolean
```

C#

```
public bool Abort  { get; set; }
```

Remarks

Restrictions: The SARowsCopiedEventArgs class is not available in the .NET Compact Framework 2.0.

See also

- "SARowsCopiedEventArgs class" on page 397
- "SARowsCopiedEventArgs members" on page 397

RowsCopied property

Gets the number of rows copied during the current bulk-copy operation.

Syntax

Visual Basic

```
Public Readonly Property RowsCopied As Long
```

C#

```
public long RowsCopied  { get;}
```

Remarks

Restrictions: The SARowsCopiedEventArgs class is not available in the .NET Compact Framework 2.0.

See also

- "SARowsCopiedEventArgs class" on page 397
- "SARowsCopiedEventArgs members" on page 397

SARowsCopiedEventHandler delegate

Represents the method that handles the SABulkCopy.SARowsCopied event of an SABulkCopy.

Syntax

Visual Basic

```
Public Delegate Sub SARowsCopiedEventHandler( _
   ByVal sender As Object, _
   ByVal rowsCopiedEventArgs As SARowsCopiedEventArgs _
)
```

C#

```
public delegate void SARowsCopiedEventHandler(
   object sender,
   SARowsCopiedEventArgs rowsCopiedEventArgs
);
```

Remarks

Restrictions: The SARowsCopiedEventHandler delegate is not available in the .NET Compact Framework 2.0.

See also

- "SABulkCopy class" on page 164

SARowUpdatedEventArgs class

Provides data for the RowUpdated event. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SARowUpdatedEventArgs
  Inherits RowUpdatedEventArgs
```

C#

```
public sealed class SARowUpdatedEventArgs : RowUpdatedEventArgs
```

See also

- "SARowUpdatedEventArgs members" on page 400

SARowUpdatedEventArgs members

Public constructors

Member name	Description
SARowUpdatedEventArgs constructor	Initializes a new instance of the SARowUpdatedEventArgs class.

Public properties

Member name	Description
Command property	Gets the SACommand that is executed when DataAdapter.Update is called.

Member name	Description
Errors (inherited from RowUpdatedEventArgs)	Gets any errors generated by the .NET Framework data provider when the RowUpdatedEventArgs.Command was executed.
RecordsAffected property	Returns the number of rows changed, inserted, or deleted by execution of the SQL statement.
Row (inherited from RowUpdatedEventArgs)	Gets the DataRow sent through an DbDataAdapter.Update.
RowCount (inherited from RowUpdatedEventArgs)	Gets the number of rows processed in a batch of updated records.
StatementType (inherited from RowUpdatedEventArgs)	Gets the type of SQL statement executed.
Status (inherited from RowUpdatedEventArgs)	Gets the UpdateStatus of the RowUpdatedEventArgs.Command.
TableMapping (inherited from RowUpdatedEventArgs)	Gets the DataTableMapping sent through an DbDataAdapter.Update.

Public methods

Member name	Description
CopyToRows (inherited from RowUpdatedEventArgs)	Copies references to the modified rows into the provided array.

See also

- "SARowUpdatedEventArgs class" on page 400

SARowUpdatedEventArgs constructor

Initializes a new instance of the SARowUpdatedEventArgs class.

Syntax

Visual Basic

```
Public Sub New( _
   ByVal row As DataRow, _
   ByVal command As IDbCommand, _
   ByVal statementType As StatementType, _
   ByVal tableMapping As DataTableMapping _
)
```

C#

```
public  SARowUpdatedEventArgs(
  DataRow row,
  IDbCommand command,
  StatementType statementType,
  DataTableMapping tableMapping
);
```

Parameters

- **row** The DataRow sent through an Update.
- **command** The IDbCommand executed when Update is called.
- **statementType** One of the StatementType values that specifies the type of query executed.
- **tableMapping** The DataTableMapping sent through an Update.

See also

- "SARowUpdatedEventArgs class" on page 400
- "SARowUpdatedEventArgs members" on page 400

Command property

Gets the SACommand that is executed when DataAdapter.Update is called.

Syntax

Visual Basic

```
Public Readonly Property Command As SACommand
```

C#

```
public SACommand Command  { get;}
```

See also

- "SARowUpdatedEventArgs class" on page 400
- "SARowUpdatedEventArgs members" on page 400

RecordsAffected property

Returns the number of rows changed, inserted, or deleted by execution of the SQL statement.

Syntax

Visual Basic

```
Public Readonly Property RecordsAffected As Integer
```


C#

```
public int RecordsAffected  { get;}
```

Property value

The number of rows changed, inserted, or deleted; 0 if no rows were affected or the statement failed; and -1 for SELECT statements.

See also

- "SARowUpdatedEventArgs class" on page 400
- "SARowUpdatedEventArgs members" on page 400

SARowUpdatedEventHandler delegate

Represents the method that handles the RowUpdated event of an SADataAdapter.

Syntax

Visual Basic

```
Public Delegate Sub SARowUpdatedEventHandler( _
   ByVal sender As Object, _
   ByVal e As SARowUpdatedEventArgs _
)
```

C#

```
public delegate void SARowUpdatedEventHandler(
   object sender,
   SARowUpdatedEventArgs e
);
```

SARowUpdatingEventArgs class

Provides data for the RowUpdating event. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SARowUpdatingEventArgs
  Inherits RowUpdatingEventArgs
```

C#

```
public sealed class SARowUpdatingEventArgs : RowUpdatingEventArgs
```

See also

- "SARowUpdatingEventArgs members" on page 404

SARowUpdatingEventArgs members

Public constructors

Member name	Description
SARowUpdatingEventArgs constructor	Initializes a new instance of the SARowUpdatingEventArgs class.

Public properties

Member name	Description
Command property	Specifies the SACommand to execute when performing the Update.
Errors (inherited from RowUpdatingEventArgs)	Gets any errors generated by the .NET Framework data provider when the RowUpdatedEventArgs.Command executes.
Row (inherited from RowUpdatingEventArgs)	Gets the DataRow that will be sent to the server as part of an insert, update, or delete operation.
StatementType (inherited from RowUpdatingEventArgs)	Gets the type of SQL statement to execute.
Status (inherited from RowUpdatingEventArgs)	Gets or sets the UpdateStatus of the RowUpdatedEventArgs.Command.
TableMapping (inherited from RowUpdatingEventArgs)	Gets the DataTableMapping to send through the DbDataAdapter.Update.

See also

- "SARowUpdatingEventArgs class" on page 403

SARowUpdatingEventArgs constructor

Initializes a new instance of the SARowUpdatingEventArgs class.

Syntax

Visual Basic

```
Public Sub New( _
   ByVal row As DataRow, _
   ByVal command As IDbCommand, _
   ByVal statementType As StatementType, _
   ByVal tableMapping As DataTableMapping _
)
```


C#

```
public  SARowUpdatingEventArgs(
  DataRow row,
  IDbCommand command,
  StatementType statementType,
  DataTableMapping tableMapping
);
```

Parameters

- **row** The DataRow to update.
- **command** The IDbCommand to execute during update.
- **statementType** One of the StatementType values that specifies the type of query executed.
- **tableMapping** The DataTableMapping sent through an Update.

See also

- "SARowUpdatingEventArgs class" on page 403
- "SARowUpdatingEventArgs members" on page 404

Command property

Specifies the SACommand to execute when performing the Update.

Syntax

Visual Basic

```
Public Property Command As SACommand
```

C#

```
public SACommand Command  { get; set; }
```

See also

- "SARowUpdatingEventArgs class" on page 403
- "SARowUpdatingEventArgs members" on page 404

SARowUpdatingEventHandler delegate

Represents the method that handles the RowUpdating event of an SADataAdapter.

Syntax

Visual Basic

```
Public Delegate Sub SARowUpdatingEventHandler( _
  ByVal sender As Object, _
  ByVal e As SARowUpdatingEventArgs _
)
```

C#

```
public delegate void SARowUpdatingEventHandler(
   object sender,
   SARowUpdatingEventArgs e
);
```

SATcpOptionsBuilder class

Provides a simple way to create and manage the TCP options portion of connection strings used by the SAConnection object. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SATcpOptionsBuilder
  Inherits SAConnectionStringBuilderBase
```

C#

```
public sealed class SATcpOptionsBuilder : SAConnectionstring BuilderBase
```

Remarks

Restrictions: The SATcpOptionsBuilder class is not available in the .NET Compact Framework 2.0.

See also

- "SATcpOptionsBuilder members" on page 406
- "SAConnection class" on page 231

SATcpOptionsBuilder members

Public constructors

Member name	Description
SATcpOptionsBuilder constructors	Initializes a new instance of the "SATcpOptionsBuilder class" on page 406.

Public properties

Member name	Description
Broadcast property	Gets or sets the Broadcast option.
BroadcastListener property	Gets or sets the BroadcastListener option.

Member name	Description
BrowsableConnectionString (inherited from DbConnectionStringBuilder)	Gets or sets a value that indicates whether the DbConnectionStringBuilder.ConnectionString is visible in Visual Studio designers.
ClientPort property	Gets or sets the ClientPort option.
ConnectionString (inherited from DbConnectionStringBuilder)	Gets or sets the connection string associated with the DbConnectionStringBuilder.
Count (inherited from DbConnectionStringBuilder)	Gets the current number of keys that are contained within the DbConnectionStringBuilder.ConnectionString.
DoBroadcast property	Gets or sets the DoBroadcast option.
Host property	Gets or sets the Host option.
IPV6 property	Gets or sets the IPV6 option.
IsFixedSize (inherited from DbConnectionStringBuilder)	Gets a value that indicates whether the DbConnectionStringBuilder has a fixed size.
IsReadOnly (inherited from DbConnectionStringBuilder)	Gets a value that indicates whether the DbConnectionStringBuilder is read-only.
Item property (inherited from SAConnectionStringBuilderBase)	Gets or sets the value of the connection keyword.
Keys property (inherited from SAConnectionStringBuilderBase)	Gets an System.Collections.ICollection that contains the keys in the SAConnectionStringBuilder.
LDAP property	Gets or sets the LDAP option.
LocalOnly property	Gets or sets the LocalOnly option.
MyIP property	Gets or sets the MyIP option.
ReceiveBufferSize property	Gets or sets the ReceiveBufferSize option.
SendBufferSize property	Gets or sets the Send BufferSize option.
ServerPort property	Gets or sets the ServerPort option.
TDS property	Gets or sets the TDS option.

Member name	Description
Timeout property	Gets or sets the Timeout option.
Values (inherited from DbConnectionStringBuilder)	Gets an ICollection that contains the values in the DbConnectionStringBuilder.
VerifyServerName property	Gets or sets the VerifyServerName option.

Public methods

Member name	Description
Add (inherited from DbConnectionStringBuilder)	Adds an entry with the specified key and value into the DbConnectionStringBuilder.
Clear (inherited from DbConnectionStringBuilder)	Clears the contents of the DbConnectionStringBuilder instance.
ContainsKey method (inherited from SAConnectionStringBuilderBase)	Determines whether the SAConnectionStringBuilder object contains a specific keyword.
EquivalentTo (inherited from DbConnectionStringBuilder)	Compares the connection information in this DbConnectionStringBuilder object with the connection information in the supplied object.
GetKeyword method (inherited from SAConnectionStringBuilderBase)	Gets the keyword for specified SAConnectionStringBuilder property.
GetUseLongNameAsKeyword method (inherited from SAConnectionStringBuilderBase)	Gets a boolean values that indicates whether long connection parameter names are used in the connection string.
Remove method (inherited from SAConnectionStringBuilderBase)	Removes the entry with the specified key from the SAConnectionStringBuilder instance.
SetUseLongNameAsKeyword method (inherited from SAConnectionStringBuilderBase)	Sets a boolean value that indicates whether long connection parameter names are used in the connection string. Long connection parameter names are used by default.
ShouldSerialize method (inherited from SAConnectionStringBuilderBase)	Indicates whether the specified key exists in this SAConnectionStringBuilder instance.
ToString method	Converts the TcpOptionsBuilder object to a string representation.

Member name	Description
TryGetValue method (inherited from SAConnectionStringBuilderBase)	Retrieves a value corresponding to the supplied key from this SAConnectionStringBuilder.

See also

- "SATcpOptionsBuilder class" on page 406
- "SAConnection class" on page 231

SATcpOptionsBuilder constructors

Initializes a new instance of the "SATcpOptionsBuilder class" on page 406.

SATcpOptionsBuilder() constructor

Initializes an SATcpOptionsBuilder object.

Syntax

Visual Basic

Public Sub **New()**

C#

public **SATcpOptionsBuilder();**

Remarks

Restrictions: The SATcpOptionsBuilder class is not available in the .NET Compact Framework 2.0.

Example

The following statement initializes an SATcpOptionsBuilder object.

```
SATcpOptionsBuilder options = new SATcpOptionsBuilder( );
```

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406
- "SATcpOptionsBuilder constructors" on page 409

SATcpOptionsBuilder(String) constructor

Initializes an SATcpOptionsBuilder object.

Syntax

Visual Basic

```
Public Sub New( _
  ByVal options As String _
)
```

C#

```
public  SATcpOptionsBuilder(
  string  options
);
```

Parameters

- **options** A SQL Anywhere TCP connection parameter options string.

  For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Remarks

Restrictions: The SATcpOptionsBuilder class is not available in the .NET Compact Framework 2.0.

Example

The following statement initializes an SATcpOptionsBuilder object.

```
SATcpOptionsBuilder options = new SATcpOptionsBuilder( );
```

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406
- "SATcpOptionsBuilder constructors" on page 409

Broadcast property

Gets or sets the Broadcast option.

Syntax

Visual Basic

```
Public Property Broadcast As String
```

C#

```
public string  Broadcast  { get; set; }
```

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

BroadcastListener property

Gets or sets the BroadcastListener option.

Syntax

Visual Basic

Public Property **BroadcastListener** As String

C#

public string **BroadcastListener** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

ClientPort property

Gets or sets the ClientPort option.

Syntax

Visual Basic

Public Property **ClientPort** As String

C#

public string **ClientPort** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

DoBroadcast property

Gets or sets the DoBroadcast option.

Syntax

Visual Basic

Public Property **DoBroadcast** As String

C#

public string **DoBroadcast** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

Host property

Gets or sets the Host option.

Syntax

Visual Basic

```
Public Property Host As String
```

C#

```
public string  Host  { get; set; }
```

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

IPV6 property

Gets or sets the IPV6 option.

Syntax

Visual Basic

```
Public Property IPV6 As String
```

C#

```
public string  IPV6  { get; set; }
```

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

LDAP property

Gets or sets the LDAP option.

Syntax

Visual Basic

```
Public Property LDAP As String
```


C#

public string **LDAP** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

LocalOnly property

Gets or sets the LocalOnly option.

Syntax

Visual Basic

Public Property **LocalOnly** As String

C#

public string **LocalOnly** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

MyIP property

Gets or sets the MyIP option.

Syntax

Visual Basic

Public Property **MyIP** As String

C#

public string **MyIP** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

ReceiveBufferSize property

Gets or sets the ReceiveBufferSize option.

Syntax

Visual Basic

Public Property **ReceiveBufferSize** As Integer

C#

public int **ReceiveBufferSize** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

SendBufferSize property

Gets or sets the Send BufferSize option.

Syntax

Visual Basic

Public Property **SendBufferSize** As Integer

C#

public int **SendBufferSize** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

ServerPort property

Gets or sets the ServerPort option.

Syntax

Visual Basic

Public Property **ServerPort** As String

C#

public string **ServerPort** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

TDS property

Gets or sets the TDS option.

Syntax

Visual Basic

Public Property **TDS** As String

C#

public string **TDS** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

Timeout property

Gets or sets the Timeout option.

Syntax

Visual Basic

Public Property **Timeout** As Integer

C#

public int **Timeout** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

VerifyServerName property

Gets or sets the VerifyServerName option.

Syntax

Visual Basic

Public Property **VerifyServerName** As String

C#

public string **VerifyServerName** { get; set; }

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

ToString method

Converts the TcpOptionsBuilder object to a string representation.

Syntax

Visual Basic

```
Public Overrides Function ToString() As String
```

C#

```
public override string  ToString();
```

Return value

The options string being built.

See also

- "SATcpOptionsBuilder class" on page 406
- "SATcpOptionsBuilder members" on page 406

SATransaction class

Represents a SQL transaction. This class cannot be inherited.

Syntax

Visual Basic

```
Public NotInheritable Class SATransaction
  Inherits DbTransaction
```

C#

```
public sealed class SATransaction : DbTransaction
```

Remarks

There is no constructor for SATransaction. To obtain an SATransaction object, use one of the BeginTransaction methods. To associate a command with a transaction, use the SACommand.Transaction property.

For more information, see "Transaction processing" on page 134 and "Inserting, updating, and deleting rows using the SACommand object" on page 118.

See also

- "SATransaction members" on page 417
- "BeginTransaction() method" on page 239
- "BeginTransaction(SAIsolationLevel) method" on page 240
- "Transaction property" on page 199

SATransaction members

Public properties

Member name	Description
Connection property	The SAConnection object associated with the transaction, or a null reference (Nothing in Visual Basic) if the transaction is no longer valid.
IsolationLevel property	Specifies the isolation level for this transaction.
SAIsolationLevel property	Specifies the isolation level for this transaction.

Public methods

Member name	Description
Commit method	Commits the database transaction.
Dispose (inherited from DbTransaction)	Releases the unmanaged resources used by the DbTransaction.
Rollback methods	Rolls back a transaction from a pending state.
Save method	Creates a savepoint in the transaction that can be used to roll back a portion of the transaction, and specifies the savepoint name.

See also

- "SATransaction class" on page 416
- "BeginTransaction() method" on page 239
- "BeginTransaction(SAIsolationLevel) method" on page 240
- "Transaction property" on page 199

Connection property

The SAConnection object associated with the transaction, or a null reference (Nothing in Visual Basic) if the transaction is no longer valid.

Syntax

Visual Basic

Public Readonly Property **Connection** As SAConnection

C#

public SAConnection **Connection** { get;}

Remarks

A single application can have multiple database connections, each with zero or more transactions. This property enables you to determine the connection object associated with a particular transaction created by BeginTransaction.

See also

- "SATransaction class" on page 416
- "SATransaction members" on page 417

IsolationLevel property

Specifies the isolation level for this transaction.

Syntax

Visual Basic

Public Overrides Readonly Property **IsolationLevel** As IsolationLevel

C#

public override IsolationLevel **IsolationLevel** { get;}

Property value

The isolation level for this transaction. This can be one of:

- ReadCommitted
- ReadUncommitted
- RepeatableRead
- Serializable
- Snapshot
- ReadOnlySnapshot
- StatementSnapshot

The default is ReadCommitted.

See also

- "SATransaction class" on page 416
- "SATransaction members" on page 417

SAIsolationLevel property

Specifies the isolation level for this transaction.

Syntax

Visual Basic

Public Readonly Property **SAIsolationLevel** As SAIsolationLevel

C#

public SAIsolationLevel **SAIsolationLevel** { get;}

Property value

The IsolationLevel for this transaction. This can be one of:

- Chaos
- Read ReadCommitted
- ReadOnlySnapshot
- ReadUncommitted
- RepeatableRead
- Serializable
- Snapshot
- StatementSnapshot
- Unspecified

The default is ReadCommitted.

Remarks

Parallel transactions are not supported. Therefore, the IsolationLevel applies to the entire transaction.

See also

- "SATransaction class" on page 416
- "SATransaction members" on page 417

Commit method

Commits the database transaction.

Syntax

Visual Basic

Public Overrides Sub **Commit()**

C#

public override void **Commit();**

See also

- "SATransaction class" on page 416
- "SATransaction members" on page 417

Rollback methods

Rolls back a transaction from a pending state.

Rollback() method

Rolls back a transaction from a pending state.

Syntax

Visual Basic

```
Public Overrides Sub Rollback()
```

C#

```
public override void Rollback();
```

Remarks

The transaction can only be rolled back from a pending state (after BeginTransaction has been called, but before Commit is called).

See also

- "SATransaction class" on page 416
- "SATransaction members" on page 417
- "Rollback methods" on page 420

Rollback(String) method

Rolls back a transaction from a pending state.

Syntax

Visual Basic

```
Public Sub Rollback( _
  ByVal savePoint As String _
)
```

C#

```
public void Rollback(
  string  savePoint
);
```


Parameters

- **savePoint** The name of the savepoint to roll back to.

Remarks

The transaction can only be rolled back from a pending state (after BeginTransaction has been called, but before Commit is called).

See also

- "SATransaction class" on page 416
- "SATransaction members" on page 417
- "Rollback methods" on page 420

Save method

Creates a savepoint in the transaction that can be used to roll back a portion of the transaction, and specifies the savepoint name.

Syntax

Visual Basic

```
Public Sub Save( _
  ByVal savePoint As String _
)
```

C#

```
public void Save(
  string  savePoint
);
```

Parameters

- **savePoint** The name of the savepoint to which to roll back.

See also

- "SATransaction class" on page 416
- "SATransaction members" on page 417

iAnywhere.SQLAnywhere.Server namespace (.NET 2.0)

SAServerSideConnection class

Initializes a new instance of the "SAServerSideConnection class" on page 422.

Syntax

Visual Basic

Public NotInheritable Class **SAServerSideConnection**

C#

public sealed class **SAServerSideConnection**

See also

- "SAServerSideConnection members" on page 422

SAServerSideConnection members

Public constructors

Member name	Description
SAServerSideConnection constructor	

Public properties

Member name	Description
Connection property	

See also

- "SAServerSideConnection class" on page 422

SAServerSideConnection constructor

Syntax

Visual Basic

Public Sub **New()**

C#

public **SAServerSideConnection();**

See also

- “SAServerSideConnection class” on page 422
- “SAServerSideConnection members” on page 422

Connection property

Syntax

Visual Basic

Public Shared Readonly Property **Connection** As SAConnection

C#

public const SAConnection **Connection** { get;}

See also

- “SAServerSideConnection class” on page 422
- “SAServerSideConnection members” on page 422

CHAPTER 11

SQL Anywhere OLE DB and ADO APIs

Contents

Introduction to OLE DB

OLE DB is a data access model from Microsoft. It uses the Component Object Model (COM) interfaces and, unlike ODBC, OLE DB does not assume that the data source uses a SQL query processor.

SQL Anywhere includes an **OLE DB provider** named **SAOLEDB**. This provider is available for current Windows and Windows Mobile platforms.

You can also access SQL Anywhere using the Microsoft OLE DB Provider for ODBC (MSDASQL), together with the SQL Anywhere ODBC driver.

Using the SQL Anywhere OLE DB provider brings several benefits:

- Some features, such as updating through a cursor, are not available using the OLE DB/ODBC bridge.
- If you use the SQL Anywhere OLE DB provider, ODBC is not required in your deployment.
- MSDASQL allows OLE DB clients to work with any ODBC driver, but does not guarantee that you can use the full range of functionality of each ODBC driver. Using the SQL Anywhere provider, you can get full access to SQL Anywhere features from OLE DB programming environments.

Supported platforms

The SQL Anywhere OLE DB provider is designed to work with OLE DB 2.5 and later. For Windows Mobile and its successors, the OLE DB provider is designed for ADOCE 3.1 and later.

ADOCE is the Microsoft ADO for Windows CE SDK and provides database functionality for applications developed with the Windows CE Toolkits for Visual Basic 5.0 and Visual Basic 6.0.

For a list of supported platforms, see SQL Anywhere Supported Platforms and Engineering Support Status.

Distributed transactions

The OLE DB driver can be used as a resource manager in a distributed transaction environment.

For more information, see “Three-tier computing and distributed transactions” on page 63.

ADO programming with SQL Anywhere

ADO (ActiveX Data Objects) is a data access object model exposed through an Automation interface, which allows client applications to discover the methods and properties of objects at runtime without any prior knowledge of the object. Automation allows scripting languages like Visual Basic to use a standard data access object model. ADO uses OLE DB to provide data access.

Using the SQL Anywhere OLE DB provider, you get full access to SQL Anywhere features from an ADO programming environment.

This section describes how to perform basic tasks while using ADO from Visual Basic. It is not a complete guide to programming using ADO.

Code samples from this section can be found in the following files:

Development tool	Sample
Microsoft Visual Basic	*samples-dir\SQLAnywhere\VBSampler\vbsampler.sln*
Microsoft eMbedded Visual Basic 3.0	*samples-dir\SQLAnywhere\ADOCE\OLEDB_PocketPC.ebp*

For information about programming in ADO, see your development tool documentation.

Connecting to a database with the Connection object

This section describes a simple Visual Basic routine that connects to a database.

Sample code

You can try this routine by placing a command button named Command1 on a form, and pasting the routine into its Click event. Run the program and click the button to connect and then disconnect.

```
Private Sub cmdTestConnection_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdTestConnection.Click

    ' Declare variables
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim cAffected As Integer

    On Error GoTo HandleError

    ' Establish the connection
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 11 Demo"
    myConn.Open()
    MsgBox("Connection succeeded")
    myConn.Close()
    Exit Sub
```

```
HandleError:
    MsgBox(ErrorToString(Err.Number))
    Exit Sub
End Sub
```

Notes

The sample carries out the following tasks:

- It declares the variables used in the routine.
- It establishes a connection, using the SQL Anywhere OLE DB provider, to the sample database.
- It uses a Command object to execute a simple statement, which displays a message in the database server messages window.
- It closes the connection.

When the SAOLEDB provider is installed, it registers itself. This registration process includes making registry entries in the COM section of the registry, so that ADO can locate the DLL when the SAOLEDB provider is called. If you change the location of your DLL, you must re-register it.

To register the OLE DB provider

1. Open a command prompt.
2. Change to the directory where the OLE DB provider is installed.
3. Enter the following commands to register the provider:

```
regsvr32 dboledb11.dll
regsvr32 dboledba11.dll
```

For more information about connecting to a database using OLE DB, see "Connecting to a database using OLE DB" [*SQL Anywhere Server - Database Administration*].

Executing statements with the Command object

This section describes a simple routine that sends a simple SQL statement to the database.

Sample code

You can try this routine by placing a command button named Command2 on a form, and pasting the routine into its Click event. Run the program and click the button to connect, display a message in the database server messages window, and then disconnect.

```
Private Sub cmdUpdate_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdUpdate.Click

    ' Declare variables
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim cAffected As Integer

    On Error GoTo HandleError
```



```
    ' Establish the connection
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 11 Demo"
    myConn.Open()

    'Execute a command
    myCommand.CommandText = _
        "UPDATE Customers SET GivenName='Liz' WHERE ID=102"
    myCommand.ActiveConnection = myConn
    myCommand.Execute(cAffected)
    MsgBox(CStr(cAffected) & " rows affected.",
        MsgBoxStyle.Information)

    myConn.Close()
    Exit Sub

HandleError:
    MsgBox(ErrorToString(Err.Number))
    Exit Sub
End Sub
```

Notes

After establishing a connection, the example code creates a Command object, sets its CommandText property to an update statement, and sets its ActiveConnection property to the current connection. It then executes the update statement and displays the number of rows affected by the update in a window.

In this example, the update is sent to the database and committed as soon as it is executed.

For information about using transactions within ADO, see "Using transactions" on page 432.

You can also perform updates through a cursor.

For more information, see "Updating data through a cursor" on page 431.

Querying the database with the Recordset object

The ADO Recordset object represents the result set of a query. You can use it to view data from a database.

Sample code

You can try this routine by placing a command button named cmdQuery on a form and pasting the routine into its Click event. Run the program and click the button to connect, display a message in the database server messages window, execute a query and display the first few rows in windows, and then disconnect.

```
Private Sub cmdQuery_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdQuery.Click

    ' Declare variables
    Dim i As Integer
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim myRS As New ADODB.Recordset
```

```
        On Error GoTo ErrorHandler

        ' Establish the connection
        myConn.Provider = "SAOLEDB"
        myConn.ConnectionString = _
            "Data Source=SQL Anywhere 11 Demo"
        myConn.CursorLocation = _
            ADODB.CursorLocationEnum.adUseServer
        myConn.Mode = _
            ADODB.ConnectModeEnum.adModeReadWrite
        myConn.IsolationLevel = _
            ADODB.IsolationLevelEnum.adXactCursorStability
        myConn.Open()

        'Execute a query
        myRS = New ADODB.Recordset
        myRS.CacheSize = 50
        myRS.let_Source("SELECT * FROM Customers")
        myRS.let_ActiveConnection(myConn)
        myRS.CursorType = ADODB.CursorTypeEnum.adOpenKeyset
        myRS.LockType = ADODB.LockTypeEnum.adLockOptimistic
        myRS.Open()

        'Scroll through the first few results
        myRS.MoveFirst()
        For i = 1 To 5
            MsgBox(myRS.Fields("CompanyName").Value,
                MsgBoxStyle.Information)
            myRS.MoveNext()
        Next

        myRS.Close()
        myConn.Close()
        Exit Sub

    ErrorHandler:
        MsgBox(ErrorToString(Err.Number))
        Exit Sub
    End Sub
```

Notes

The Recordset object in this example holds the results from a query on the Customers table. The For loop scrolls through the first several rows and displays the CompanyName value for each row.

This is a simple example of using a cursor from ADO.

For more advanced examples of using a cursor from ADO, see "Working with the Recordset object" on page 430.

Working with the Recordset object

When working with SQL Anywhere, the ADO Recordset represents a cursor. You can choose the type of cursor by declaring a CursorType property of the Recordset object before you open the Recordset. The choice of cursor type controls the actions you can take on the Recordset and has performance implications.

Cursor types

ADO has its own naming convention for cursor types. The set of cursor types supported by SQL Anywhere is described in "Cursor properties" on page 37.

The available cursor types, the corresponding cursor type constants, and the SQL Anywhere types they are equivalent to, are as follows:

ADO cursor type	ADO constant	SQL Anywhere type
Dynamic cursor	adOpenDynamic	Dynamic scroll cursor
Keyset cursor	adOpenKeyset	Scroll cursor
Static cursor	adOpenStatic	Insensitive cursor
Forward only	adOpenForwardOnly	No-scroll cursor

For information about choosing a cursor type that is suitable for your application, see "Choosing cursor types" on page 37.

Sample code

The following code sets the cursor type for an ADO Recordset object:

```
Dim myRS As New ADODB.Recordset
myRS.CursorType = ADODB.CursorTypeEnum.adOpenDynamic
```

Updating data through a cursor

The SQL Anywhere OLE DB provider lets you update a result set through a cursor. This capability is not available through the MSDASQL provider.

Updating record sets

You can update the database through a Recordset.

```
Private Sub cmdUpdateThroughCursor_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdUpdateThroughCursor.Click

    ' Declare variables
    Dim i As Integer
    Dim myConn As New ADODB.Connection
    Dim myRS As New ADODB.Recordset
    Dim SQLString As String

    On Error GoTo HandleError

    ' Connect
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 11 Demo"
    myConn.Open()
    myConn.BeginTrans()
```

```
        SQLString = "SELECT * FROM Customers"
        myRS.Open(SQLString, myConn, _
            ADODB.CursorTypeEnum.adOpenDynamic, _
            ADODB.LockTypeEnum.adLockBatchOptimistic)

        If myRS.BOF And myRS.EOF Then
            MsgBox("Recordset is empty!", 16, "Empty Recordset")
        Else
            MsgBox("Cursor type: " & CStr(myRS.CursorType), _
                MsgBoxStyle.Information)
            myRS.MoveFirst()
            For i = 1 To 3
                MsgBox("Row: " & CStr(myRS.Fields("ID").Value), _
                    MsgBoxStyle.Information)
                If i = 2 Then
                    myRS.Update("City", "Toronto")
                    myRS.UpdateBatch()
                End If
                myRS.MoveNext()
            Next i
            myRS.Close()
        End If
        myConn.CommitTrans()
        myConn.Close()
        Exit Sub

    HandleError:
        MsgBox(ErrorToString(Err.Number))
        Exit Sub

    End Sub
```

Notes

If you use the adLockBatchOptimistic setting on the Recordset, the myRS.Update method does not make any changes to the database itself. Instead, it updates a local copy of the Recordset.

The myRS.UpdateBatch method makes the update to the database server, but does not commit it, because it is inside a transaction. If an UpdateBatch method was invoked outside a transaction, the change would be committed.

The myConn.CommitTrans method commits the changes. The Recordset object has been closed by this time, so there is no issue of whether the local copy of the data is changed or not.

Using transactions

By default, any change you make to the database using ADO is committed as soon as it is executed. This includes explicit updates, as well as the UpdateBatch method on a Recordset. However, the previous section illustrated that you can use the BeginTrans and RollbackTrans or CommitTrans methods on the Connection object to use transactions.

The transaction isolation level is set as a property of the Connection object. The IsolationLevel property can take on one of the following values:

ADO isolation level	Constant	SQL Anywhere level
Unspecified	adXactUnspecified	Not applicable. Set to 0
Chaos	adXactChaos	Unsupported. Set to 0
Browse	adXactBrowse	0
Read uncommitted	adXactReadUncommitted	0
Cursor stability	adXactCursorStability	1
Read committed	adXactReadCommitted	1
Repeatable read	adXactRepeatableRead	2
Isolated	adXactIsolated	3
Serializable	adXactSerializable	3
Snapshot	2097152	4
Statement snapshot	4194304	5
Readonly statement snapshot	8388608	6

For more information about isolation levels, see “Isolation levels and consistency” [*SQL Anywhere Server - SQL Usage*].

Setting up a Microsoft Linked Server using OLE DB

A Microsoft Linked Server can be created that uses the SQL Anywhere OLE DB provider to obtain access to a SQL Anywhere database. SQL queries can be issued using either Microsoft's 4-part table referencing syntax or Microsoft's OPENQUERY SQL function. An example of the 4-part syntax follows.

```
SELECT * FROM SADATABASE..GROUPO.Customers
```

In this example, SADATABASE is the name of the Linked Server, GROUPO is the table owner in the SQL Anywhere database, and Customers is the table name in the SQL Anywhere database. The catalog name is omitted (as indicated by two consecutive dots) since catalog names are not a feature of SQL Anywhere databases.

The other form uses Microsoft's OPENQUERY function.

```
SELECT * FROM OPENQUERY( SADATABASE, 'SELECT * FROM Customers' )
```

In the OPENQUERY syntax, the second SELECT statement ('SELECT * FROM Customers') is passed to the SQL Anywhere server for execution.

To set up a Linked Server that uses the SQL Anywhere OLE DB provider, a few steps must be followed.

To set up a Linked Server

1. Fill in the General page.

   The Linked Server field on the General page should contain a Linked Server name (like SADATABASE used above). The Other Data Source option should be selected, and SQL Anywhere OLE DB Provider should be selected from the list. The Product Name field should contain an ODBC data source name (for example, SQL Anywhere 11 Demo). The Provider String field can contain additional connection parameters such as user ID and password (for example, uid=DBA;pwd=sql). Other fields, such as Data Source, on the General page should be left empty.

2. Select the Allow Inprocess provider option.

   The technique for doing this varies with different versions of Microsoft SQL Server. In SQL Server 2000, there is a Provider Options button that takes you to the page where you can select this option. In SQL Server 2005, there is a global Allow Inprocess checkbox when you right-click the SAOLEDB provider in the Linked Servers/Providers tree view and choose Properties. If the InProcess option is not selected, queries fail.

3. Select the RPC and RPC Out options.

   The technique for doing this varies with different versions of Microsoft SQL Server. In SQL Server 2000, there are two checkboxes that must be selected for these two options. These check boxes are found on the Server Options page. In SQL Server 2005, the options are True/False settings. Make sure that they are set True. The Remote Procedure Call (RPC) options must be selected if you want to execute stored procedure/function calls in a SQL Anywhere database and pass parameters in and out successfully.

Supported OLE DB interfaces

The OLE DB API consists of a set of interfaces. The following table describes the support for each interface in the SQL Anywhere OLE DB driver.

Interface	Purpose	Limitations
IAccessor	Define bindings between client memory and data store values.	DBACCESSOR_PASSBYREF not supported. DBACCESSOR_OPTIMIZED not supported.
IAlterIndex IAlterTable	Alter tables, indexes, and columns.	Not supported.
IChapteredRowset	A chaptered rowset allows rows of a rowset to be accessed in separate chapters.	Not supported. SQL Anywhere does not support chaptered rowsets.
IColumnsInfo	Get simple information about the columns in a rowset.	Not supported on Windows Mobile.
IColumnsRowset	Get information about optional metadata columns in a rowset, and get a rowset of column metadata.	Not supported on Windows Mobile.
ICommand	Execute SQL statements.	Does not support calling. ICommandProperties: GetProperties with DBPROPSET_PROPERTIESINERROR to find properties that could not have been set.
ICommandPersist	Persist the state of a command object (but not any active rowsets). These persistent command objects can subsequently be enumerated using the PROCEDURES or VIEWS rowset.	Not supported on Windows Mobile.
ICommandPrepare	Prepare commands.	Not supported on Windows Mobile.
ICommandProperties	Set Rowset properties for rowsets created by a command. Most commonly used to specify the interfaces the rowset should support.	Supported.

Interface	Purpose	Limitations
ICommandText	Set the SQL statement text for ICommand.	Only the DBGUID_DEFAULT SQL dialect is supported.
ICommandWithParameters	Set or get parameter information for a command.	No support for parameters stored as vectors of scalar values. No support for BLOB parameters. Not on Windows Mobile.
IConvertType		Supported. Limited on Windows Mobile.
IDBAsynchNotify IDBAsyncStatus	Asynchronous processing. Notify client of events in the asynchronous processing of data source initialization, populating rowsets, and so on.	Not supported.
IDBCreateCommand	Create commands from a session.	Supported.
IDBCreateSession	Create a session from a data source object.	Supported.
IDBDataSourceAdmin	Create/destroy/modify data source objects, which are COM objects used by clients. This interface is not used to manage data stores (databases).	Not supported.
IDBInfo	Find information about keywords unique to this provider (that is, to find non-standard SQL keywords). Also, find information about literals, special characters used in text matching queries, and other literal information.	Not supported on Windows Mobile.
IDBInitialize	Initialize data source objects and enumerators.	Not supported on Windows Mobile.
IDBProperties	Manage properties on a data source object or enumerator.	Not supported on Windows Mobile.

Interface	Purpose	Limitations
IDBSchemaRowset	Get information about system tables, in a standard form (a rowset).	Not supported on Windows Mobile.
IErrorInfo IErrorLookup IErrorRecords	ActiveX error object support.	Not supported on Windows Mobile.
IGetDataSource	Returns an interface pointer to the session's data source object.	Supported.
IIndexDefinition	Create or drop indexes in the data store.	Not supported.
IMultipleResults	Retrieve multiple results (rowsets or row counts) from a command.	Supported.
IOpenRowset	Non-SQL way to access a database table by its name.	Supported. Opening a table by its name is supported, not by a GUID.
IParentRowset	Access chaptered/hierarchical rowsets.	Not supported.
IRowset	Access rowsets.	Supported.
IRowsetChange	Allow changes to rowset data, reflected back to the data store. InsertRow/SetData for BLOBs are not implemented.	Not supported on Windows Mobile.
IRowsetChapterMember	Access chaptered/hierarchical rowsets.	Not supported.
IRowsetCurrentIndex	Dynamically change the index for a rowset.	Not supported.
IRowsetFind	Find a row within a rowset matching a specified value.	Not supported.
IRowsetIdentity	Compare row handles.	Not supported.
IRowsetIndex	Access database indexes.	Not supported.

Interface	Purpose	Limitations
IRowsetInfo	Find information about rowset properties or to find the object that created the rowset.	Not supported on Windows Mobile.
IRowsetLocate	Position on rows of a rowset, using bookmarks.	Not supported on Windows Mobile.
IRowsetNotify	Provides a COM callback interface for rowset events.	Supported.
IRowsetRefresh	Get the latest value of data that is visible to a transaction.	Not supported.
IRowsetResynch	Old OLEDB 1.x interface, superseded by IRowsetRefresh.	Not supported.
IRowsetScroll	Scroll through rowset to fetch row data.	Not supported.
IRowsetUpdate	Delay changes to rowset data until Update is called.	Supported. Not on Windows Mobile.
IRowsetView	Use views on an existing rowset.	Not supported.
ISequentialStream	Retrieve a BLOB column.	Supported for reading only. No support for SetData with this interface. Not on Windows Mobile.
ISessionProperties	Get session property information.	Supported.
ISourcesRowset	Get a rowset of data source objects and enumerators.	Not supported on Windows Mobile.
ISQLErrorInfo ISupportErrorInfo	ActiveX error object support.	Optional on Windows Mobile.
ITableDefinition ITableDefinitionWithConstraints	Create, drop, and alter tables, with constraints.	Not supported on Windows Mobile.
ITransaction	Commit or abort transactions.	Not all the flags are supported. Not on Windows Mobile.

Interface	Purpose	Limitations
ITransactionJoin	Support distributed transactions.	Not all the flags are supported. Not on Windows Mobile.
ITransactionLocal	Handle transactions on a session. Not all the flags are supported.	Not supported on Windows Mobile.
ITransactionOptions	Get or set options on a transaction.	Not supported on Windows Mobile.
IViewChapter	Work with views on an existing rowset, specifically to apply post-processing filters/sorting on rows.	Not supported.
IViewFilter	Restrict contents of a rowset to rows matching a set of conditions.	Not supported.
IViewRowset	Restrict contents of a rowset to rows matching a set of conditions, when opening a rowset.	Not supported.
IViewSort	Apply sort order to a view.	Not supported.

CHAPTER 12

SQL Anywhere ODBC API

Contents

Introduction to ODBC

The **Open Database Connectivity** (**ODBC**) interface is an application programming interface defined by Microsoft Corporation as a standard interface to database management systems on Windows operating systems. ODBC is a call-based interface.

To write ODBC applications for SQL Anywhere, you need:

- SQL Anywhere.
- A C compiler capable of creating programs for your environment.
- The Microsoft ODBC Software Development Kit. This is available on the Microsoft Developer Network, and provides documentation and additional tools for testing ODBC applications.

Supported platforms

SQL Anywhere supports the ODBC API on Unix and Windows Mobile, in addition to Windows. Having multi-platform ODBC support makes portable database application development much easier.

For information on enlisting the ODBC driver in distributed transactions, see "Three-tier computing and distributed transactions" on page 63.

See also

- Microsoft Open Database Connectivity (ODBC)

> **Note**
> Some application development tools that already have ODBC support provide their own programming interface that hides the ODBC interface. The SQL Anywhere documentation does not describe how to use those tools.

ODBC conformance

SQL Anywhere provides support for ODBC 3.5, which is supplied as part of the Microsoft Data Access Kit 2.7.

Levels of ODBC support

ODBC features are arranged according to level of conformance. Features are either **Core**, **Level 1**, or **Level 2**, with Level 2 being the most complete level of ODBC support. These features are listed in the Microsoft ODBC Programmer's Reference.

Features supported by SQL Anywhere

SQL Anywhere supports the ODBC 3.5 specification as follows:

- **Core conformance** SQL Anywhere supports all Core level features.
- **Level 1 conformance** SQL Anywhere supports all Level 1 features, except for asynchronous execution of ODBC functions.

SQL Anywhere supports multiple threads sharing a single connection. The requests from the different threads are serialized by SQL Anywhere.

- **Level 2 conformance** SQL Anywhere supports all Level 2 features, except for the following ones:
  - Three part names of tables and views. This is not applicable for SQL Anywhere.
  - Asynchronous execution of ODBC functions for specified individual statements.
  - Ability to time out login requests and SQL queries.

ODBC backward compatibility

Applications developed using older versions of ODBC continue to work with SQL Anywhere and the newer ODBC driver manager. The new ODBC features are not provided for older applications.

The ODBC driver manager

Microsoft Windows includes an ODBC driver manager. For Unix, an ODBC driver manager is supplied with SQL Anywhere.

Building ODBC applications

This section describes how to compile and link simple ODBC applications.

Including the ODBC header file

Every C source file that calls ODBC functions must include a platform-specific ODBC header file. Each platform-specific header file includes the main ODBC header file *odbc.h*, which defines all the functions, data types, and constant definitions required to write an ODBC program.

To include the ODBC header file in a C source file

1. Add an include line referencing the appropriate platform-specific header file to your source file. The lines to use are as follows:

Operating system	Include line
Windows	#include "ntodbc.h"
Unix	#include "unixodbc.h"
Windows Mobile	#include "ntodbc.h"

2. Add the directory containing the header file to the include path for your compiler.

   Both the platform-specific header files and *odbc.h* are installed in the *SDK\Include* subdirectory of your SQL Anywhere installation directory.

3. When building ODBC applications for Unix, you might have to define the macro "UNIX" for 32-bit applications or "UNIX64" for 64-bit applications in order to obtain the correct data alignment and sizes. This step is not required if you are using one of the following supported compilers:

   - GNU C/C++ compiler on any of our supported platforms
   - Intel C/C++ compiler for Linux (icc)
   - SunPro C/C++ compiler for Linux or Solaris
   - VisualAge C/C++ compiler for AIX
   - C/C++ compiler (cc/aCC) for HP-UX

Linking ODBC applications on Windows

This section does not apply to Windows Mobile.

For Windows Mobile information, see “Linking ODBC applications on Windows Mobile” on page 445.

When linking your application, you must link against the appropriate import library file to have access to the ODBC functions. The import library defines entry points for the ODBC driver manager *odbc32.dll*. The driver manager in turn loads the SQL Anywhere ODBC driver *dbodbc11.dll*.

To link an ODBC application (Windows)

1. Add the directory containing the platform-specific import library to the list of library directories in your LIB environment variable.

   Typically, the import library is stored under the *Lib* directory structure of the Microsoft platform SDK:

Operating system	Import library
Windows (32-bit)	*Lib\odbc32.lib*
Windows (64-bit)	*Lib\x64\odbc32.lib*

   Here is an example for a command prompt.

   ```
   set LIB=%LIB%;C:\mssdk\v6.1\lib
   ```

2. To compile and link a simple ODBC application using the Microsoft compile and link tool, you can issue a single command from the command prompt. The following example compiles and links the application stored in *odbc.c*.

   ```
   cl odbc.c /Ic:\sa11\SDK\Include odbc32.lib
   ```

Linking ODBC applications on Windows Mobile

On Windows Mobile operating systems, there is no ODBC driver manager. The import library (*dbodbc11.lib*) defines entry points directly into the SQL Anywhere ODBC driver *dbodbc11.dll*. This file is located in the *SDK\Lib\CE\Arm.50* subdirectory of your SQL Anywhere installation.

Since there is no ODBC driver manager for Windows Mobile, you must specify the location of the SQL Anywhere ODBC driver DLL for the DRIVER= parameter in the connection string supplied to the SQLDriverConnect function. The following is an example.

```
szConnStrIn = "driver=ospath\\dbodbc11.dll;dbf=\\samples-dir\\demo.db"
```

Here, *ospath* is the full path to the Windows directory on the Windows Mobile device. For example:

```
\\Windows
```

To link an ODBC application (Windows Mobile)

- Add the directory containing the platform-specific import library to the list of library directories.

For a list of supported versions of Windows Mobile, see the SQL Anywhere for PC Platforms table in SQL Anywhere Supported Platforms and Engineering Support Status.

The sample program (*odbc_sample.cpp*) uses a File Data Source (FileDSN connection parameter) called *SQL Anywhere 11 Demo.dsn*. This file placed in the root directory of your Windows Mobile device when you install SQL Anywhere for Windows Mobile to your device. You can create file data sources on your desktop system with the ODBC Data Source Administrator, but they must be set up for your desktop environment and then edited to match the Windows Mobile environment. After appropriate edits, you can copy them to your Windows Mobile device.

For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

Windows Mobile and Unicode

SQL Anywhere uses an encoding known as UTF-8, a multibyte character encoding that can be used to encode Unicode.

The SQL Anywhere ODBC driver supports either ASCII (8-bit) strings or Unicode code (wide character) strings. The UNICODE macro controls whether ODBC functions expect ASCII or Unicode strings. If your application must be built with the UNICODE macro defined, but you want to use the ASCII ODBC functions, then the SQL_NOUNICODEMAP macro must also be defined.

The sample file *samples-dir\SQLAnywhere\C\odbc.c* illustrates how to use the Unicode ODBC features.

Linking ODBC applications on Unix

An ODBC driver manager is included with SQL Anywhere and there are third party driver managers available. This section describes how to build ODBC applications that do not use an ODBC driver manager.

ODBC driver

The ODBC driver is a shared object or shared library. Separate versions of the SQL Anywhere ODBC driver are supplied for single-threaded and multi-threaded applications. A generic SQL Anywhere ODBC driver is supplied that will detect the threading model in use and direct calls to the appropriate single-threaded or multi-threaded library.

The ODBC drivers are the following files:

Operating system	Threading model	ODBC driver
(all Unix except Mac OS X and HP-UX)	Generic	*libdbodbc11.so (libdbodbc11.so.1)*
(all Unix except Mac OS X and HP-UX)	Single threaded	*libdbodbc11_n.so (libdbodbc11_n.so.1)*
(all Unix except Mac OS X and HP-UX)	Multi-threaded	*libdbodbc11_r.so (libdbodbc11_r.so.1)*
HP-UX	Generic	*libdbodbc11.sl (libdbodbc11.sl.1)*
HP-UX	Single threaded	*libdbodbc11_n.sl (libdbodbc11_n.sl.1)*
HP-UX	Multi-threaded	*libdbodbc11_r.sl (libdbodbc11_r.sl.1)*
Mac OS X	Generic	*libdbodbc11.dylib*
Mac OS X	Single threaded	*libdbodbc11_n.dylib*

Operating system	Threading model	ODBC driver
Mac OS X	Multi-threaded	*libdbodbc11_r.dylib*

The libraries are installed as symbolic links to the shared library with a version number (shown in parentheses).

In addition, the following bundles are also provided for Mac OS X:

Operating system	Threading model	ODBC driver
Mac OS X	Single threaded	*dbodbc11.bundle*
Mac OS X	Multi-threaded	*dbodbc11_r.bundle*

To link an ODBC application (Unix)

1. Link your application against the generic ODBC driver *libdbodbc11*.
2. When deploying your application, ensure that the appropriate (or all) ODBC driver versions (non-threaded or threaded) are available in the user's library path.

Data source information

If SQL Anywhere does not detect the presence of an ODBC driver manager, it uses the system information file for data source information. See "Using ODBC data sources on Unix" [*SQL Anywhere Server - Database Administration*].

Using an ODBC driver manager on Unix

SQL Anywhere includes an ODBC driver manager for Unix. The *libdbodm11* shared object can be used on all supported Unix platforms as an ODBC driver manager. The iAnywhere ODBC driver manager can be used to load any version 3.0 or later ODBC driver. The driver manager will not perform mappings between ODBC 1.0/2.0 calls and ODBC 3.x calls; therefore, applications using the iAnywhere ODBC driver manager must restrict their use of the ODBC feature set to version 3.0 and later. Also, the iAnywhere ODBC driver manager can be used by both threaded and non-threaded applications.

The iAnywhere ODBC driver manager can perform tracing of ODBC calls for any given connection. To turn on tracing, a user can use the TraceLevel and TraceLog directives. These directives can be part of a connection string (in the case where SQLDriverConnect is being used) or within a DSN entry. The TraceLog is a log file where the traced output for the connection goes while the TraceLevel is the amount of tracing information wanted. The trace levels are:

- **NONE** No tracing information is printed.
- **MINIMAL** Routine name and parameters are included in the output.
- **LOW** In addition to the above, return values are included in the output.
- **MEDIUM** In addition to the above, the date and time of execution are included in the output.
- **HIGH** In addition to the above, parameter types are included in the output.

Also, third-party ODBC driver managers for Unix are available. Consult the documentation that accompanies these driver managers for information on their use.

ODBC samples

Several ODBC samples are included with SQL Anywhere. You can find the samples in the *samples-dir\SQLAnywhere* subdirectories.

The samples in directories starting with ODBC illustrate separate and simple ODBC tasks, such as connecting to a database and executing statements. A complete sample ODBC program is supplied in *samples-dir\SQLAnywhere\C\odbc.c*. This program performs the same actions as the embedded SQL dynamic cursor example program that is in the same directory.

For a description of the associated embedded SQL program, see "Sample embedded SQL programs" on page 514.

Building the sample ODBC program

The ODBC sample program is located in *samples-dir\SQLAnywhere\C*. A batch file (shell script for Unix) is included that can be used to compile and link all the sample applications located in this folder.

To build the sample ODBC program

1. Open a command prompt and change directory to the *samples-dir\SQLAnywhere\C* directory.
2. Run the *build.bat* or *build64.bat* batch file.

   For x64 platform builds, you may need to set up the correct environment for compiling and linking. Here is an example that builds the sample programs for an x64 platform.

   ```
   set mssdk=c:\MSSDK\v6.1
   build64
   ```

To build the sample ODBC program for Unix

1. Open a command shell and change directory to the *samples-dir\SQLAnywhere\C* directory.
2. Run the *build.sh* shell script.

Running the sample ODBC program

To run the ODBC sample

1. Start the program:
   - For 32-bit Windows, run the file *samples-dir\SQLAnywhere\C\odbcwin.exe*.
   - For 64-bit Windows, run the file *samples-dir\SQLAnywhere\C\odbcx64.exe*.
   - For Unix, run the file *samples-dir\SQLAnywhere\C\odbc*.
2. Choose a table:
   - Choose one of the tables in the sample database. For example, you can enter **Customers** or **Employees**.

ODBC handles

ODBC applications use a small set of **handles** to define basic features such as database connections and SQL statements. A handle is a 32-bit value.

The following handles are used in essentially all ODBC applications:

- **Environment** The environment handle provides a global context in which to access data. Every ODBC application must allocate exactly one environment handle upon starting, and must free it at the end.

  The following code illustrates how to allocate an environment handle:

  ```
  SQLHENV env;
  SQLRETURN rc;
  rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL
     _NULL_HANDLE, &env );
  ```

- **Connection** A connection is specified by an ODBC driver and a data source. An application can have several connections associated with its environment. Allocating a connection handle does not establish a connection; a connection handle must be allocated first and then used when the connection is established.

  The following code illustrates how to allocate a connection handle:

  ```
  SQLHDBC  dbc;
  SQLRETURN rc;
  rc = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
  ```

- **Statement** A statement handle provides access to a SQL statement and any information associated with it, such as result sets and parameters. Each connection can have several statements. Statements are used both for cursor operations (fetching data) and for single statement execution (for example, INSERT, UPDATE, and DELETE).

  The following code illustrates how to allocate a statement handle:

  ```
  SQLHSTMT stmt;
  SQLRETURN rc;
  rc = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
  ```

Allocating ODBC handles

The handle types required for ODBC programs are as follows:

Item	Handle type
Environment	SQLHENV
Connection	SQLHDBC
Statement	SQLHSTMT

Item	Handle type
Descriptor	SQLHDESC

To use an ODBC handle

1. Call the SQLAllocHandle function.

   SQLAllocHandle takes the following parameters:

   - an identifier for the type of item being allocated
   - the handle of the parent item
   - a pointer to the location of the handle to be allocated

   For a full description, see SQLAllocHandle in the Microsoft *ODBC Programmer's Reference*.
2. Use the handle in subsequent function calls.
3. Free the object using SQLFreeHandle.

   SQLFreeHandle takes the following parameters:

   - an identifier for the type of item being freed
   - the handle of the item being freed

   For a full description, see SQLFreeHandle in the Microsoft ODBC Programmer's Reference.

Example

The following code fragment allocates and frees an environment handle:

```
SQLHENV env;
SQLRETURN retcode;
retcode = SQLAllocHandle(
      SQL_HANDLE_ENV,
      SQL_NULL_HANDLE,
      &env );
if( retcode == SQL_SUCCESS
    || retcode == SQL_SUCCESS_WITH_INFO ) {
  // success: application Code here
}
SQLFreeHandle( SQL_HANDLE_ENV, env );
```

For more information about return codes and error handling, see “Handling errors” on page 473.

A first ODBC example

The following is a simple ODBC program that connects to the SQL Anywhere sample database and immediately disconnects.

You can find this sample in *samples-dir\SQLAnywhere\ODBCConnect\odbcconnect.cpp*.

```
#include <stdio.h>
#include "ntodbc.h"

int main(int argc, char* argv[])
```

```
{
  SQLHENV  env;
  SQLHDBC  dbc;
  SQLRETURN retcode;

  retcode = SQLAllocHandle( SQL_HANDLE_ENV,
            SQL_NULL_HANDLE,
            &env );
  if (retcode == SQL_SUCCESS
      || retcode == SQL_SUCCESS_WITH_INFO) {
    printf( "env allocated\n" );
    /* Set the ODBC version environment attribute */
    retcode = SQLSetEnvAttr( env,
                  SQL_ATTR_ODBC_VERSION,
                  (void*)SQL_OV_ODBC3, 0);
      retcode = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
      if (retcode == SQL_SUCCESS
           || retcode == SQL_SUCCESS_WITH_INFO) {
        printf( "dbc allocated\n" );
        retcode = SQLConnect( dbc,
          (SQLCHAR*) "SQL Anywhere 11 Demo", SQL_NTS,
          (SQLCHAR* ) "DBA", SQL_NTS,
          (SQLCHAR*) "sql", SQL_NTS );
        if (retcode == SQL_SUCCESS
           || retcode == SQL_SUCCESS_WITH_INFO) {
          printf( "Successfully connected\n" );
        }
        SQLDisconnect( dbc );
      }
      SQLFreeHandle( SQL_HANDLE_DBC, dbc );
    }
    SQLFreeHandle( SQL_HANDLE_ENV, env );
  return 0;
}
```


Choosing an ODBC connection function

ODBC supplies a set of connection functions. Which one you use depends on how you expect your application to be deployed and used:

- **SQLConnect** The simplest connection function.

  SQLConnect takes a data source name and optional user ID and password. You may want to use SQLConnect if you hard-code a data source name into your application.

  For more information, see SQLConnect in the Microsoft *ODBC Programmer's Reference*.

- **SQLDriverConnect** Connects to a data source using a connection string.

  SQLDriverConnect allows the application to use SQL Anywhere-specific connection information that is external to the data source. Also, you can use SQLDriverConnect to request that the SQL Anywhere driver prompt for connection information.

  SQLDriverConnect can also be used to connect without specifying a data source.

  For more information, see SQLDriverConnect in the Microsoft *ODBC Programmer's Reference*.

- **SQLBrowseConnect** Connects to a data source using a connection string, like SQLDriverConnect.

  SQLBrowseConnect allows your application to build its own windows to prompt for connection information and to browse for data sources used by a particular driver (in this case the SQL Anywhere driver).

  For more information, see SQLBrowseConnect in the Microsoft *ODBC Programmer's Reference*.

For a complete list of connection parameters that can be used in connection strings, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Establishing a connection

Your application must establish a connection before it can perform any database operations.

To establish an ODBC connection

1. Allocate an ODBC environment.

   For example:

   ```
   SQLHENV   env;
   SQLRETURN retcode;
   retcode = SQLAllocHandle( SQL_HANDLE_ENV,
     SQL_NULL_HANDLE, &env );
   ```

2. Declare the ODBC version.

   By declaring that the application follows ODBC version 3, SQLSTATE values and some other version-dependent features are set to the proper behavior. For example:

```
retcode = SQLSetEnvAttr( env,
  SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
```

3. If necessary, assemble the data source or connection string.

   Depending on your application, you may have a hard-coded data source or connection string, or you may store it externally for greater flexibility.

4. Allocate an ODBC connection item.

   For example:

   ```
   retcode = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
   ```

5. Set any connection attributes that must be set before connecting.

   Some connection attributes must be set before establishing a connection or after establishing a connection, while others can be set either before or after. The SQL_AUTOCOMMIT attribute is one that can be set before or after:

   ```
   retcode = SQLSetConnectAttr( dbc,
      SQL_AUTOCOMMIT,
      (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
   ```

   For more information, see "Setting connection attributes" on page 454.

6. Call the ODBC connection function.

   For example:

   ```
   if (retcode == SQL_SUCCESS
       || retcode == SQL_SUCCESS_WITH_INFO) {
      printf( "dbc allocated\n" );
      retcode = SQLConnect( dbc,
         (SQLCHAR*) "SQL Anywhere 11 Demo", SQL_NTS,
         (SQLCHAR* ) "DBA", SQL_NTS,
         (SQLCHAR*) "sql", SQL_NTS );
      if (retcode == SQL_SUCCESS
            || retcode == SQL_SUCCESS_WITH_INFO){
      // successfully connected.
   ```

You can find a complete sample in *samples-dir\SQLAnywhere\ODBCConnect\odbcconnect.cpp*.

Notes

- **SQL_NTS** Every string passed to ODBC has a corresponding length. If the length is unknown, you can pass SQL_NTS indicating that it is a **Null Terminated String** whose end is marked by the null character (\0).
- **SQLSetConnectAttr** By default, ODBC operates in autocommit mode. This mode is turned off by setting SQL_AUTOCOMMIT to false.

  For more information, see "Setting connection attributes" on page 454.

Setting connection attributes

You use the SQLSetConnectAttr function to control details of the connection. For example, the following statement turns off ODBC autocommit behavior.


```
retcode = SQLSetConnectAttr( dbc, SQL_AUTOCOMMIT,
   (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
```

For more information including a list of connection attributes, see SQLSetConnectAttr in the Microsoft *ODBC Programmer's Reference*.

Many aspects of the connection can be controlled through the connection parameters. For information, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Getting connection attributes

You use the SQLGetConnectAttr function to get details of the connection. For example, the following statement returns the connection state.

```
retcode = SQLGetConnectAttr( dbc, SQL_ATTR_CONNECTION_DEAD,
   (SQLPOINTER)&closed, SQL_IS_INTEGER, 0 );
```

When using the SQLGetConnectAttr function to get the SQL_ATTR_CONNECTION_DEAD attribute, the value SQL_CD_TRUE is returned if the connection has been dropped even if no request has been sent to the server since the connection was dropped. Determining if the connection has been dropped is done without making a request to the server, and the dropped connection is detected within a few seconds. The connection can be dropped for a number of reasons, such as an idle timeout.

For more information including a list of connection attributes, see SQLGetConnectAttr in the Microsoft *ODBC Programmer's Reference*.

Threads and connections in ODBC applications

You can develop multi-threaded ODBC applications for SQL Anywhere. It is recommended that you use a separate connection for each thread.

You can use a single connection for multiple threads. However, the database server does not allow more than one active request for any one connection at a time. If one thread executes a statement that takes a long time, all other threads must wait until the request is complete.

SQL Anywhere connection attributes

The SQL Anywhere ODBC driver supports some extended connection attributes.

- **SA_REGISTER_MESSAGE_CALLBACK** Messages can be sent to the client application from the server using the SQL MESSAGE statement. A message handler routine can created to intercept these messages. The message handler callback prototype is as follows:

  void SQL_CALLBACK *message_handler***(**
  SQLHDBC *sqlany_dbc***,**
  unsigned char *msg_type***,**
  long *code***,**
  unsigned short *length***,**
  char * *message*
  );

  The following possible values for *msg_type* are defined in *sqldef.h*.

  - **MESSAGE_TYPE_INFO** The message type was INFO.
  - **MESSAGE_TYPE_WARNING** The message type was WARNING.
  - **MESSAGE_TYPE_ACTION** The message type was ACTION.
  - **MESSAGE_TYPE_STATUS** The message type was STATUS.

  A SQLCODE associated with the message may be provided in*code*. When not available, the*code* parameter value is 0.

  The length of the message is contained in *length*.

  A pointer to the message is contained in *message*. Note that *message* is not null-terminated. Your application must be designed to handle this. The following is an example.

  ```
  memcpy( mybuff, msg, len );
  mybuff[ len ] = '\0';
  ```

  To register the message handler in ODBC, call the SQLSetConnectAttr function as follows:

  ```
  rc = SQLSetConnectAttr(
     hdbc,
     SA_REGISTER_MESSAGE_CALLBACK,
     (SQLPOINTER) &message_handler, SQL_IS_POINTER );
  ```

  To unregister the message handler in ODBC, call the SQLSetConnectAttr function as follows:

  ```
  rc = SQLSetConnectAttr(
     hdbc,
     SA_REGISTER_MESSAGE_CALLBACK,
     NULL, SQL_IS_POINTER );
  ```

- **SA_GET_MESSAGE_CALLBACK_PARM** To retrieve the value of the SQLHDBC connection handle that will be passed to message handler callback routine, use SQLGetConnectAttr with the SA_GET_MESSAGE_CALLBACK_PARM parameter.

  ```
  SQLHDBC callback_hdbc = NULL;
  rc = SQLGetConnectAttr(
     hdbc,
  ```



```
SA_GET_MESSAGE_CALLBACK_PARM,
(SQLPOINTER) &callback_hdbc, 0, 0 );
```

The returned value will be the same as the parameter value that is passed to the message handler callback routine.

- **SA_REGISTER_VALIDATE_FILE_TRANSFER_CALLBACK** This is used to register a file transfer validation callback function. Before allowing any transfer to take place, the ODBC driver will invoke the validation callback, if it exists. If the client data transfer is being requested during the execution of indirect statements such as from within a stored procedure, the ODBC driver will not allow a transfer unless the client application has registered a validation callback. The conditions under which a validation call is made are described more fully below.

  The callback prototype is as follows:

  int SQL_CALLBACK *file_transfer_callback*(
  void * *sqlca*,
  char * *file_name*,
  int *is_write*
  );

  The *file_name* parameter is the name of the file to be read or written. The *is_write* parameter is 0 if a read is requested (transfer from the client to the server), and non-zero for a write. The callback function should return 0 if the file transfer is not allowed, non-zero otherwise.

  For data security, the server tracks the origin of statements requesting a file transfer. The server determines if the statement was received directly from the client application. When initiating the transfer of data from the client, the server sends the information about the origin of the statement to the client software. On its part, the embedded SQL client library allows unconditional transfer of data only if the data transfer is being requested due to the execution of a statement sent directly by the client application. Otherwise, the application must have registered the validation callback described above, in the absence of which the transfer is denied and the statement fails with an error. Note that if the client statement invokes a stored procedure already existing in the database, then the execution of the stored procedure itself is considered not to have been for a client initiated statement. However, if the client application explicitly creates a temporary stored procedure then the execution of the stored procedure results in the server treating the procedure as having been client initiated. Similarly, if the client application executes a batch statement, then the execution of the batch statement is considered as being done directly by the client application.

- **SA_SQL_ATTR_TXN_ISOLATION** This is used to set an extended transaction isolation level. The following example sets a snapshot isolation level:

  ```
  SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
  SQLSetConnectAttr( dbc, SQL_ATTR_TXN_ISOLATION,
        SA_SQL_TXN_SNAPSHOT, SQL_IS_UINTEGER );
  ```

  For more information, see “Choosing ODBC transaction isolation level” on page 462.

Executing SQL statements

ODBC includes several functions for executing SQL statements:

- **Direct execution** SQL Anywhere parses the SQL statement, prepares an access plan, and executes the statement. Parsing and access plan preparation are called **preparing** the statement.
- **Prepared execution** The statement preparation is carried out separately from the execution. For statements that are to be executed repeatedly, this avoids repeated preparation and so improves performance.

  For more information, see "Executing prepared statements" on page 460.

Executing statements directly

The SQLExecDirect function prepares and executes a SQL statement. The statement may include parameters.

The following code fragment illustrates how to execute a statement without parameters. The SQLExecDirect function takes a statement handle, a SQL string, and a length or termination indicator, which in this case is a null-terminated string indicator.

The procedure described in this section is straightforward but inflexible. The application cannot take any input from the user to modify the statement. For a more flexible method of constructing statements, see "Executing statements with bound parameters" on page 459.

To execute a SQL statement in an ODBC application

1. Allocate a handle for the statement using **SQLAllocHandle**.

   For example, the following statement allocates a handle of type `SQL_HANDLE_STMT` with name `stmt`, on a connection with handle `dbc`:

   ```
   SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
   ```

2. Call the SQLExecDirect function to execute the statement:

   For example, the following lines declare a statement and execute it. The declaration of `deletestmt` would usually occur at the beginning of the function:

   ```
   SQLCHAR deletestmt[ STMT_LEN ] =
     "DELETE FROM Departments WHERE DepartmentID = 201";
   SQLExecDirect( stmt, deletestmt, SQL_NTS) ;
   ```

For a complete sample with error checking, see *samples-dir\SQLAnywhere\ODBCExecute\odbcexecute.cpp*.

For more information about SQLExecDirect, see SQLExecDirect in the Microsoft *ODBC Programmer's Reference*.

Executing statements with bound parameters

This section describes how to construct and execute a SQL statement, using bound parameters to set values for statement parameters at runtime.

To execute a SQL statement with bound parameters in an ODBC application

1. Allocate a handle for the statement using SQLAllocHandle.

   For example, the following statement allocates a handle of type `SQL_HANDLE_STMT` with name `stmt`, on a connection with handle `dbc`:

   ```
   SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
   ```

2. Bind parameters for the statement using SQLBindParameter.

   For example, the following lines declare variables to hold the values for the department ID, department name, and manager ID, as well as for the statement string itself. They then bind parameters to the first, second, and third parameters of a statement executed using the stmt statement handle.

   ```
   #defined DEPT_NAME_LEN 40
   SQLLEN cbDeptID = 0,
      cbDeptName = SQL_NTS, cbManagerID = 0;
   SQLCHAR deptName[ DEPT_NAME_LEN + 1 ];
   SQLSMALLINT deptID, managerID;
   SQLCHAR insertstmt[ STMT_LEN ] =
     "INSERT INTO Departments "
     "( DepartmentID, DepartmentName, DepartmentHeadID )"
     "VALUES (?, ?, ?)";
   SQLBindParameter( stmt, 1, SQL_PARAM_INPUT,
       SQL_C_SSHORT, SQL_INTEGER, 0, 0,
       &deptID, 0, &cbDeptID);
   SQLBindParameter( stmt, 2, SQL_PARAM_INPUT,
       SQL_C_CHAR, SQL_CHAR, DEPT_NAME_LEN, 0,
       deptName, 0,&cbDeptName);
   SQLBindParameter( stmt, 3, SQL_PARAM_INPUT,
       SQL_C_SSHORT, SQL_INTEGER, 0, 0,
       &managerID, 0, &cbManagerID);
   ```

3. Assign values to the parameters.

   For example, the following lines assign values to the parameters for the fragment of step 2.

   ```
   deptID = 201;
   strcpy( (char * ) deptName, "Sales East" );
   managerID = 902;
   ```

   Commonly, these variables would be set in response to user action.

4. Execute the statement using SQLExecDirect.

   For example, the following line executes the statement string held in `insertstmt` on the statement handle `stmt`.

   ```
   SQLExecDirect( stmt, insertstmt, SQL_NTS) ;
   ```

Bind parameters are also used with prepared statements to provide performance benefits for statements that are executed more than once. For more information, see “Executing prepared statements” on page 460.

The above code fragments do not include error checking. For a complete sample, including error checking, see *samples-dir\SQLAnywhere\ODBCExecute\odbcexecute.cpp*.

For more information about SQLExecDirect, see SQLExecDirect in the Microsoft *ODBC Programmer's Reference*.

Executing prepared statements

Prepared statements provide performance advantages for statements that are used repeatedly. ODBC provides a full set of functions for using prepared statements.

For an introduction to prepared statements, see "Preparing statements" on page 24.

To execute a prepared SQL statement

1. Prepare the statement using SQLPrepare.

   For example, the following code fragment illustrates how to prepare an INSERT statement:

   ```
   SQLRETURN   retcode;
   SQLHSTMT    stmt;
   retcode = SQLPrepare( stmt,
               "INSERT INTO Departments
                ( DepartmentID, DepartmentName, DepartmentHeadID )
                VALUES (?, ?, ?,)",
             SQL_NTS);
   ```

   In this example:

   - **retcode** Holds a return code that should be tested for success or failure of the operation.
   - **stmt** Provides a handle to the statement so that it can be referenced later.
   - **?** The question marks are placeholders for statement parameters.

2. Set statement parameter values using SQLBindParameter.

   For example, the following function call sets the value of the DepartmentID variable:

   ```
   SQLBindParameter( stmt,
                     1,
                     SQL_PARAM_INPUT,
                     SQL_C_SSHORT,
                     SQL_INTEGER,
                     0,
                     0,
                     &sDeptID,
                     0,
                     &cbDeptID);
   ```

   In this example:

   - **stmt** is the statement handle.
   - **1** indicates that this call sets the value of the first placeholder.
   - **SQL_PARAM_INPUT** indicates that the parameter is an input statement.
   - **SQL_C_SHORT** indicates the C data type being used in the application.

- **SQL_INTEGER** indicates SQL data type being used in the database.

  The next two parameters indicate the column precision and the number of decimal digits: both zero for integers.
- **&sDeptID** is a pointer to a buffer for the parameter value.
- **0** indicates the length of the buffer, in bytes.
- **&cbDeptID** is a pointer to a buffer for the length of the parameter value.

3. Bind the other two parameters and assign values to sDeptId.
4. Execute the statement:

   ```
   retcode = SQLExecute( stmt);
   ```

   Steps 2 to 4 can be carried out multiple times.
5. Drop the statement.

   Dropping the statement frees resources associated with the statement itself. You drop statements using SQLFreeHandle.

For a complete sample, including error checking, see *samples-dir\SQLAnywhere\ODBCPrepare \odbcprepare.cpp*.

For more information about SQLPrepare, see SQLPrepare in the Microsoft *ODBC Programmer's Reference*.

Working with result sets

ODBC applications use cursors to manipulate and update result sets. SQL Anywhere provides extensive support for different kinds of cursors and cursor operations.

For an introduction to cursors, see "Working with cursors" on page 30.

Choosing ODBC transaction isolation level

You can use SQLSetConnectAttr to set the transaction isolation level for a connection. The characteristics that determine the transaction isolation level that SQL Anywhere provides include the following:

- **SQL_TXN_READ_UNCOMMITTED** Set isolation level to 0. When this attribute value is set, it isolates any data read from changes by others and changes made by others cannot be seen. The re-execution of the read statement is affected by others. This does not support a repeatable read. This is the default value for isolation level.
- **SQL_TXN_READ_COMMITTED** Set isolation level to 1. When this attribute value is set, it does not isolate data read from changes by others, and changes made by others can be seen. The re-execution of the read statement is affected by others. This does not support a repeatable read.
- **SQL_TXN_REPEATABLE_READ** Set isolation level to 2. When this attribute value is set, it isolates any data read from changes by others, and changes made by others cannot be seen. The re-execution of the read statement is affected by others. This supports a repeatable read.
- **SQL_TXN_SERIALIZABLE** Set isolation level to 3. When this attribute value is set, it isolates any data read from changes by others, and changes made by others cannot be seen. The re-execution of the read statement is not affected by others. This supports a repeatable read.
- **SA_SQL_TXN_SNAPSHOT** Set isolation level to snapshot. When this attribute value is set, it provides a single view of the database for the entire transaction.
- **SA_SQL_TXN_STATEMENT_SNAPSHOT** Set isolation level to statement-snapshot. When this attribute value is set, it provides less consistency than snapshot isolation, but may be useful in cases where long running transactions result in too much space being used in the temporary file by the version store.
- **SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT** Set isolation level to readonly-statement-snapshot. When this attribute value is set, it provides somewhat less consistency than statement-snapshot isolation, but avoids the possibility of update conflicts. Therefore, it is most appropriate for porting applications originally intended to run under different isolation levels.

For more information, see SQLSetConnectAttr in the Microsoft *ODBC Programmer's Reference*.

Example

The following fragment uses a snapshot isolation level:

```
SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
SQLSetConnectAttr( dbc, SQL_ATTR_TXN_ISOLATION,
      SA_SQL_TXN_SNAPSHOT, SQL_IS_UINTEGER );
```


Choosing ODBC cursor characteristics

ODBC functions that execute statements and manipulate result sets, use cursors to perform their tasks. Applications open a cursor implicitly whenever they execute a SQLExecute or SQLExecDirect function.

For applications that move through a result set only in a forward direction and do not update the result set, cursor behavior is relatively straightforward. By default, ODBC applications request this behavior. ODBC defines a read-only, forward-only cursor, and SQL Anywhere provides a cursor optimized for performance in this case.

For a simple example of a forward-only cursor, see "Retrieving data" on page 464.

For applications that need to scroll both forward and backward through a result set, such as many graphical user interface applications, cursor behavior is more complex. What does the application when it returns to a row that has been updated by some other application? ODBC defines a variety of **scrollable cursors** to allow you to build in the behavior that suits your application. SQL Anywhere provides a full set of cursors to match the ODBC scrollable cursor types.

You set the required ODBC cursor characteristics by calling the SQLSetStmtAttr function that defines statement attributes. You must call SQLSetStmtAttr before executing a statement that creates a result set.

You can use SQLSetStmtAttr to set many cursor characteristics. The characteristics that determine the cursor type that SQL Anywhere supplies include the following:

- **SQL_ATTR_CURSOR_SCROLLABLE** Set to SQL_SCROLLABLE for a scrollable cursor and SQL_NONSCROLLABLE for a forward-only cursor. SQL_NONSCROLLABLE is the default.
- **SQL_ATTR_CONCURRENCY** Set to one of the following values:
  - **SQL_CONCUR_READ_ONLY** Disallow updates. SQL_CONCUR_READ_ONLY is the default.
  - **SQL_CONCUR_LOCK** Use the lowest level of locking sufficient to ensure that the row can be updated.
  - **SQL_CONCUR_ROWVER** Use optimistic concurrency control, comparing row versions such as SQLBase ROWID or Sybase TIMESTAMP.
  - **SQL_CONCUR_VALUES** Use optimistic concurrency control, comparing values.

For more information, see SQLSetStmtAttr in the Microsoft *ODBC Programmer's Reference*.

Example

The following fragment requests a read-only, scrollable cursor:

```
SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
SQLSetStmtAttr( stmt, SQL_ATTR_CURSOR_SCROLLABLE,
      SQL_SCROLLABLE, SQL_IS_UINTEGER );
```

Retrieving data

To retrieve rows from a database, you execute a SELECT statement using SQLExecute or SQLExecDirect. This opens a cursor on the statement.

You then use SQLFetch or SQLFetchScroll to fetch rows through the cursor. These functions fetch the next rowset of data from the result set and return data for all bound columns. Using SQLFetchScroll, rowsets can be specified at an absolute or relative position or by bookmark. SQLFetchScroll replaces the older SQLExtendedFetch from the ODBC 2.0 specification.

When an application frees the statement using SQLFreeHandle, it closes the cursor.

To fetch values from a cursor, your application can use either SQLBindCol or SQLGetData. If you use SQLBindCol, values are automatically retrieved on each fetch. If you use SQLGetData, you must call it for each column after each fetch.

SQLGetData is used to fetch values in pieces for columns such as LONG VARCHAR or LONG BINARY. As an alternative, you can set the SQL_ATTR_MAX_LENGTH statement attribute to a value large enough to hold the entire value for the column. The default value for SQL_ATTR_MAX_LENGTH is 256 KB.

The SQL Anywhere ODBC driver implements SQL_ATTR_MAX_LENGTH in a different way than intended by the ODBC specification. The intended meaning for SQL_ATTR_MAX_LENGTH is that it be used as a mechanism to truncate large fetches. This might be done for a "preview" mode where only the first part of the data is displayed. For example, instead of transmitting a 4 MB blob from the server to the client application, only the first 500 bytes of it might be transmitted (by setting SQL_ATTR_MAX_LENGTH to 500). The SQL Anywhere ODBC driver does not support this implementation.

The following code fragment opens a cursor on a query and retrieves data through the cursor. Error checking has been omitted to make the example easier to read. The fragment is taken from a complete sample, which can be found in *samples-dir\SQLAnywhere\ODBCSelect\odbcselect.cpp*.

```
SQLINTEGER cbDeptID = 0, cbDeptName = SQL_NTS, cbManagerID = 0;
SQLCHAR deptName[ DEPT_NAME_LEN + 1 ];
SQLSMALLINT deptID, managerID;
SQLHENV  env;
SQLHDBC  dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env );
SQLSetEnvAttr( env,
            SQL_ATTR_ODBC_VERSION,
            (void*)SQL_OV_ODBC3, 0);
SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
SQLConnect( dbc,
            (SQLCHAR*) "SQL Anywhere 11 Demo", SQL_NTS,
            (SQLCHAR*) "DBA", SQL_NTS,
            (SQLCHAR*) "sql", SQL_NTS );
SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
SQLBindCol( stmt, 1,
            SQL_C_SSHORT, &deptID, 0, &cbDeptID);
SQLBindCol( stmt, 2,
            SQL_C_CHAR, deptName,
            sizeof(deptName), &cbDeptName);
SQLBindCol( stmt, 3,
            SQL_C_SSHORT, &managerID, 0, &cbManagerID);
SQLExecDirect( stmt, (SQLCHAR * )
```



```
"SELECT DepartmentID, DepartmentName, DepartmentHeadID FROM Departments "
                "ORDER BY DepartmentID", SQL_NTS );
while( ( retcode = SQLFetch( stmt ) ) != SQL_NO_DATA ){
   printf( "%d %20s %d\n", deptID, deptName, managerID );
}
SQLFreeHandle( SQL_HANDLE_STMT, stmt );
SQLDisconnect( dbc );
SQLFreeHandle( SQL_HANDLE_DBC, dbc );
SQLFreeHandle( SQL_HANDLE_ENV, env );
```

The number of row positions you can fetch in a cursor is governed by the size of an integer. You can fetch rows numbered up to number 2147483646, which is one less than the value that can be held in a 32-bit integer. When using negative numbers (rows from the end) you can fetch down to one more than the largest negative value that can be held in an integer.

Data alignment requirements

When you use SQLBindCol, SQLBindParameter, or SQLGetData, a C data type is specified for the column or parameter. On certain platforms, the storage (memory) provided for each column must be properly aligned to fetch or store a value of the specified type. The following table lists memory alignment requirements for all processors except x86, x64, and PowerPC platforms. Processors such as Itanium-IA64 and ARM-based devices require memory alignment.

C Data Type	Alignment required
SQL_C_CHAR	none
SQL_C_BINARY	none
SQL_C_GUID	none
SQL_C_BIT	none
SQL_C_STINYINT	none
SQL_C_UTINYINT	none
SQL_C_TINYINT	none
SQL_C_NUMERIC	none
SQL_C_DEFAULT	none
SQL_C_SSHORT	2
SQL_C_USHORT	2
SQL_C_SHORT	2
SQL_C_DATE	2

C Data Type	Alignment required
SQL_C_TIME	2
SQL_C_TIMESTAMP	2
SQL_C_TYPE_DATE	2
SQL_C_TYPE_TIME	2
SQL_C_TYPE_TIMESTAMP	2
SQL_C_WCHAR	2 (buffer size must be a multiple of 2 on all platforms)
SQL_C_SLONG	4
SQL_C_ULONG	4
SQL_C_LONG	4
SQL_C_FLOAT	4
SQL_C_DOUBLE	8
SQL_C_SBIGINT	8
SQL_C_UBIGINT	8

The x64 platform includes Advanced Micro Devices (AMD) AMD64 processors and Intel Extended Memory 64 Technology (EM64T) processors.

Parameter size requirements

When you use an ODBC function like SQLBindCol, SQLBindParameter, or SQLGetData, some of the parameters are typed as SQLLEN or SQLULEN in the function prototype. Depending on the date of the Microsoft ODBC API Reference documentation that you are looking at, you might see the same parameters described as SQLINTEGER (long) or SQLUINTEGER (unsigned long).

SQLLEN and SQLULEN data items are 64 bits in a 64-bit ODBC application and 32-bits in a 32-bit ODBC application. SQLINTEGER and SQLUINTEGER data items are 32 bits on all platforms.

To illustrate the problem, the following ODBC function prototype was excerpted from an older copy of the Microsoft ODBC API Reference.

```
SQLRETURN SQLGetData(
     SQLHSTMT     StatementHandle,
     SQLUSMALLINT ColumnNumber,
     SQLSMALLINT  TargetType,
     SQLPOINTER   TargetValuePtr,
```



```
      SQLINTEGER   BufferLength,
      SQLINTEGER  *StrLen_or_IndPtr);
```

Compare this with the actual function prototype found in *sql.h* in Microsoft's Visual Studio version 8.

```
SQLRETURN  SQL_API SQLGetData(
    SQLHSTMT      StatementHandle,
    SQLUSMALLINT  ColumnNumber,
    SQLSMALLINT   TargetType,
    SQLPOINTER    TargetValue,
    SQLLEN        BufferLength,
    SQLLEN       *StrLen_or_Ind);
```

As you can see, the BufferLength and StrLen_or_Ind parameters are now typed as SQLLEN, not SQLINTEGER. For the 64-bit platform, these are 64-bit quantities, not 32-bit quantities as inferred from Microsoft's documentation.

To avoid issues with cross-platform compilation, SQL Anywhere provides its own ODBC header files. For Windows platforms, you should include the *ntodbc.h* header file. For Unix platforms such as Linux, you should include the *unixodbc.h* header file. Use of these header files ensures compatibility with the corresponding SQL Anywhere ODBC driver for the target platform.

The following table lists some common ODBC types that have the same or different storage sizes on 64-bit and 32-bit platforms.

ODBC API	64-bit platform	32-bit platform
SQLINTEGER	32 bits	32 bits
SQLUINTEGER	32 bits	32 bits
SQLLEN	64 bits	32 bits
SQLULEN	64 bits	32 bits
SQLSETPOSIROW	64 bits	16 bits
SQL_C_BOOKMARK	64 bits	32 bits
BOOKMARK	64 bits	32 bits

If you declare data variables and parameters incorrectly, then you may encounter incorrect software behavior.

The following table summarizes the ODBC API function prototypes that have changed since the introduction of 64-bit support. The parameters that are affected are noted. The parameter name as documented by Microsoft is shown in parentheses when it differs from the actual parameter name used in the function prototype. The parameter names are those used in Microsoft's Visual Studio version 8 header files.

ODBC API	Parameter (Documented Parameter Name)
SQLBindCol	SQLLEN BufferLength SQLLEN *Strlen_or_Ind

ODBC API	Parameter (Documented Parameter Name)
SQLBindParam	SQLULEN LengthPrecision SQLLEN *Strlen_or_Ind
SQLBindParameter	SQLULEN cbColDef (ColumnSize) SQLLEN cbValueMax (BufferLength) SQLLEN *pcbValue (Strlen_or_IndPtr)
SQLColAttribute	SQLLEN *NumericAttribute
SQLColAttributes	SQLLEN *pfDesc
SQLDescribeCol	SQLULEN *ColumnSize (ColumnSizePtr)
SQLDescribeParam	SQLULEN *pcbParamDef (ParameterSizePtr)
SQLExtendedFetch	SQLLEN irow (FetchOffset) SQLULEN *pcrow (RowCountPtr)
SQLFetchScroll	SQLLEN FetchOffset
SQLGetData	SQLLEN BufferLength SQLLEN *Strlen_or_Ind (Strlen_or_IndPtr)
SQLGetDescRec	SQLLEN *Length (LengthPtr)
SQLParamOptions	SQLULEN crow, SQLULEN *pirow
SQLPutData	SQLLEN Strlen_or_Ind
SQLRowCount	SQLLEN *RowCount (RowCountPtr)
SQLSetConnectOption	SQLULEN Value
SQLSetDescRec	SQLLEN Length SQLLEN *StringLength (StringLengthPtr) SQLLEN *Indicator (IndicatorPtr)
SQLSetParam	SQLULEN LengthPrecision SQLLEN *Strlen_or_Ind (Strlen_or_IndPtr)

ODBC API	Parameter (Documented Parameter Name)
SQLSetPos	SQLSETPOSIROW irow (RowNumber)
SQLSetScrollOptions	SQLLEN crowKeyset
SQLSetStmtOption	SQLULEN Value

Some values returned from ODBC API calls through pointers have changed to accommodate 64-bit applications. For more information, see Microsoft's ODBC 64-Bit API Changes in MDAC 2.7.

Updating and deleting rows through a cursor

The Microsoft ODBC Programmer's Reference suggests that you use SELECT ... FOR UPDATE to indicate that a query is updatable using positioned operations. You do not need to use the FOR UPDATE clause in SQL Anywhere: SELECT statements are automatically updatable as long as the following conditions are met:

- The underlying query supports updates.

  That is to say, as long as a data modification statement on the columns in the result is meaningful, then positioned data modification statements can be carried out on the cursor.

  The ansi_update_constraints database option limits the type of queries that are updatable.

  For more information, see "ansi_update_constraints option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- The cursor type supports updates.

  If you are using a read-only cursor, you cannot update the result set.

ODBC provides two alternatives for carrying out positioned updates and deletes:

- Use the SQLSetPos function.

  Depending on the parameters supplied (SQL_POSITION, SQL_REFRESH, SQL_UPDATE, SQL_DELETE) SQLSetPos sets the cursor position and allows an application to refresh data, or update, or delete data in the result set.

  This is the method to use with SQL Anywhere.

- Send positioned UPDATE and DELETE statements using SQLExecute. This method should not be used with SQL Anywhere.

Using bookmarks

ODBC provides **bookmarks**, which are values used to identify rows in a cursor. SQL Anywhere supports bookmarks for value-sensitive and insensitive cursors. For example, this means that the ODBC cursor types SQL_CURSOR_STATIC and SQL_CURSOR_KEYSET_DRIVEN support bookmarks while cursor types SQL_CURSOR_DYNAMIC and SQL_CURSOR_FORWARD_ONLY do not.

Before ODBC 3.0, a database could specify only whether it supported bookmarks or not: there was no interface to provide this information for each cursor type. There was no way for a database server to indicate for what kind of cursor bookmarks were supported. For ODBC 2 applications, SQL Anywhere returns that it does support bookmarks. There is therefore nothing to prevent you from trying to use bookmarks with dynamic cursors; however, you should not use this combination.

Calling stored procedures

This section describes how to create and call stored procedures and process the results from an ODBC application.

For a full description of stored procedures and triggers, see "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*].

Procedures and result sets

There are two types of procedures: those that return result sets and those that do not. You can use SQLNumResultCols to tell the difference: the number of result columns is zero if the procedure does not return a result set. If there is a result set, you can fetch the values using SQLFetch or SQLExtendedFetch just like any other cursor.

Parameters to procedures should be passed using parameter markers (question marks). Use SQLBindParameter to assign a storage area for each parameter marker, whether it is an INPUT, OUTPUT, or INOUT parameter.

To handle multiple result sets, ODBC must describe the currently executing cursor, not the procedure-defined result set. Therefore, ODBC does not always describe column names as defined in the RESULT clause of the stored procedure definition. To avoid this problem, you can use column aliases in your procedure result set cursor.

Example

This example creates and calls a procedure that does not return a result set. The procedure takes one INOUT parameter, and increments its value. In the example, the variable num_col has the value zero, since the procedure does not return a result set. Error checking has been omitted to make the example easier to read.

```
HDBC dbc;
HSTMT stmt;
long I;
SWORD num_col;

/* Create a procedure */
SQLAllocStmt( dbc, &stmt );
SQLExecDirect( stmt,
      "CREATE PROCEDURE Increment( INOUT a INT )" \
      " BEGIN" \
         " SET a = a + 1" \
      " END", SQL_NTS );

/* Call the procedure to increment 'I' */
I = 1;
SQLBindParameter( stmt, 1, SQL_C_LONG, SQL_INTEGER, 0,
            0, &I, NULL );
SQLExecDirect( stmt, "CALL Increment( ? )",
            SQL_NTS );
SQLNumResultCols( stmt, &num_col );
do_something( I );
```

Example

This example calls a procedure that returns a result set. In the example, the variable **num_col** will have the value 2 since the procedure returns a result set with two columns. Again, error checking has been omitted to make the example easier to read.

```
HDBC dbc;
HSTMT stmt;
SWORD num_col;
RETCODE retcode;
char ID[ 10 ];
char Surname[ 20 ];

/* Create the procedure */
SQLExecDirect( stmt,
      "CREATE PROCEDURE employees()" \
      " RESULT( ID CHAR(10), Surname CHAR(20))"\
      " BEGIN" \
      " SELECT EmployeeID, Surname FROM Employees" \
      " END", SQL_NTS );

/* Call the procedure - print the results */
SQLExecDirect( stmt, "CALL employees()", SQL_NTS );
SQLNumResultCols( stmt, &num_col );
SQLBindCol( stmt, 1, SQL_C_CHAR, &ID,
            sizeof(ID), NULL );
SQLBindCol( stmt, 2, SQL_C_CHAR, &Surname,
            sizeof(Surname), NULL );

for( ;; ) {
   retcode = SQLFetch( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) {
      retcode = SQLMoreResults( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) break;
}  else {
      do_something( ID, Surname );
   }
}
```


Handling errors

Errors in ODBC are reported using the return value from each of the ODBC function calls and either the SQLError function or the SQLGetDiagRec function. The SQLError function was used in ODBC versions up to, but not including, version 3. As of version 3 the SQLError function has been deprecated and replaced by the SQLGetDiagRec function.

Every ODBC function returns a SQLRETURN, which is one of the following status codes:

Status code	Description
SQL_SUCCESS	No error.
SQL_SUCCESS_WITH_INFO	The function completed, but a call to SQLError will indicate a warning. The most common case for this status is that a value being returned is too long for the buffer provided by the application.
SQL_ERROR	The function did not complete because of an error. Call SQLError to get more information on the problem.
SQL_INVALID_HANDLE	An invalid environment, connection, or statement handle was passed as a parameter. This often happens if a handle is used after it has been freed, or if the handle is the null pointer.
SQL_NO_DATA_FOUND	There is no information available. The most common use for this status is when fetching from a cursor; it indicates that there are no more rows in the cursor.
SQL_NEED_DATA	Data is needed for a parameter. This is an advanced feature described in the ODBC SDK documentation under SQLParamData and SQLPutData.

Every environment, connection, and statement handle can have one or more errors or warnings associated with it. Each call to SQLError or SQLGetDiagRec returns the information for one error and removes the information for that error. If you do not call SQLError or SQLGetDiagRec to remove all errors, the errors are removed on the next function call that passes the same handle as a parameter.

Each call to SQLError passes three handles for an environment, connection, and statement. The first call uses SQL_NULL_HSTMT to get the error associated with a connection. Similarly, a call with both SQL_NULL_DBC and SQL_NULL_HSTMT get any error associated with the environment handle.

Each call to SQLGetDiagRec can pass either an environment, connection or statement handle. The first call passes in a handle of type SQL_HANDLE_DBC to get the error associated with a connection. The second

call passes in a handle of type SQL_HANDLE_STMT to get the error associated with the statement that was just executed.

SQLError and SQLGetDiagRec return SQL_SUCCESS if there is an error to report (*not* SQL_ERROR), and SQL_NO_DATA_FOUND if there are no more errors to report.

Example 1

The following code fragment uses SQLError and return codes:

```
/* Declare required variables */
SQLHDBC dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
UCHAR errmsg[100];
/* Code omitted here */
retcode = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt );
if( retcode == SQL_ERROR ){
   SQLError( env, dbc, SQL_NULL_HSTMT, NULL, NULL,
         errmsg, sizeof(errmsg), NULL );
   /* Assume that print_error is defined */
   print_error( "Allocation failed", errmsg );
   return;
}

/* Delete items for order 2015 */
retcode = SQLExecDirect( stmt,
       "DELETE FROM SalesOrderItems WHERE ID=2015",
       SQL_NTS );
if( retcode == SQL_ERROR ) {
   SQLError( env, dbc, stmt, NULL, NULL,
            errmsg, sizeof(errmsg), NULL );
   /* Assume that print_error is defined */
   print_error( "Failed to delete items", errmsg );
   return;
}
```

Example 2

The following code fragment uses SQLGetDiagRec and return codes:

```
/* Declare required variables */
SQLHDBC dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
SQLSMALLINT errmsglen;
SQLINTEGER errnative;
UCHAR errmsg[255];
UCHAR errstate[5];
/* Code omitted here */
retcode = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt );
if( retcode == SQL_ERROR ){
   SQLGetDiagRec(SQL_HANDLE_DBC, dbc, 1, errstate,
       &errnative, errmsg, sizeof(errmsg), &errmsglen);
    /* Assume that print_error is defined */
   print_error( "Allocation failed",
errstate, errnative, errmsg );
   return;

}
/* Delete items for order 2015 */
retcode = SQLExecDirect( stmt,
       "DELETE FROM SalesOrderItems WHERE ID=2015",
```



```
        SQL_NTS );
if( retcode == SQL_ERROR ) {
   SQLGetDiagRec(SQL_HANDLE_STMT, stmt,
        recnum, errstate,
        &errnative, errmsg, sizeof(errmsg), &errmsglen);
   /* Assume that print_error is defined */
   print_error("Failed to delete items",
        errstate, errnative, errmsg );
   return;
}
```

CHAPTER 13

SQL Anywhere JDBC API

Contents

Introduction to JDBC

JDBC can be used both from client applications and inside the database. Java classes using JDBC provide a more powerful alternative to SQL stored procedures for incorporating programming logic into the database.

JDBC provides a SQL interface for Java applications: if you want to access relational data from Java, you do so using JDBC calls.

The phrase **client application** applies both to applications running on a user's computer and to logic running on a middle-tier application server.

The examples illustrate the distinctive features of using JDBC in SQL Anywhere. For more information about JDBC programming, see any JDBC programming book.

You can use JDBC with SQL Anywhere in the following ways:

- **JDBC on the client** Java client applications can make JDBC calls to SQL Anywhere. The connection takes place through a JDBC driver.

  SQL Anywhere supports and includes two JDBC drivers: the iAnywhere JDBC driver, which is a Type 2 JDBC driver, and the jConnect driver for pure Java applications, which is a Type 4 JDBC driver.
- **JDBC in the database** Java classes installed into a database can make JDBC calls to access and modify data in the database using an internal JDBC driver.

JDBC resources

- **Example source code** You can find source code for the examples in this chapter in the directory *samples-dir\SQLAnywhere\JDBC*.
- **JDBC Specification** You can find more on the JDBC Data Access API at Java SE Technologies - Database.
- **Required software** You need TCP/IP to use the jConnect driver.

  The jConnect driver is available at jConnect for JDBC.

  For more information about the jConnect driver and its location, see "Using the jConnect JDBC driver" on page 483.

Choosing a JDBC driver

SQL Anywhere supports the following JDBC drivers:

- **iAnywhere JDBC driver** This driver communicates with SQL Anywhere using the Command Sequence client/server protocol. Its behavior is consistent with ODBC, embedded SQL, and OLE DB applications. The iAnywhere JDBC driver is the recommended JDBC driver for connecting to SQL Anywhere databases.
- **jConnect** This driver is a 100% pure Java driver. It communicates with SQL Anywhere using the TDS client/server protocol.

  jConnect and jConnect documentation are available from jConnect for JDBC.

When choosing which driver to use, you should consider the following factors:

- **Features** Both the iAnywhere JDBC driver and jConnect 6.0.5 are JDBC 3.0 compliant. However, the iAnywhere JDBC driver provides fully-scrollable cursors when connected to a SQL Anywhere database. The jConnect JDBC driver provides fully-scrollable cursors only when connected to an Adaptive Server Enterprise database.

  The JDBC 3.0 API documentation is available at JDBC Downloads. For a summary of the iAnywhere JDBC API support, see "iAnywhere JDBC 3.0 API support" on page 505.
- **Pure Java** The jConnect driver is a pure Java solution. The iAnywhere JDBC driver requires the SQL Anywhere ODBC driver and is not a pure Java solution.
- **Performance** The iAnywhere JDBC driver provides better performance for most purposes than the jConnect driver.
- **Compatibility** The TDS protocol used by the jConnect driver is shared with Adaptive Server Enterprise. Some aspects of the driver's behavior are governed by this protocol, and are configured to be compatible with Adaptive Server Enterprise.

For information about platform availability for iAnywhere JDBC driver and jConnect, see SQL Anywhere Supported Platforms and Engineering Status.

For information about using jConnect with Windows Mobile, see "Using jConnect on Windows Mobile" [*SQL Anywhere Server - Database Administration*].

JDBC program structure

The following sequence of events typically occurs in JDBC applications:

1. **Create a Connection object**

   Calling a getConnection class method of the DriverManager class creates a Connection object, and establishes a connection with a database.
2. **Generate a Statement object**

   The Connection object generates a Statement object.
3. **Pass a SQL statement**

   A SQL statement that executes within the database environment is passed to the Statement object. If the statement is a query, this action returns a ResultSet object.

   The ResultSet object contains the data returned from the SQL statement, but exposes it one row at a time (similar to the way a cursor works).
4. **Loop over the rows of the result set**

   The next method of the ResultSet object performs two actions:

   - The current row (the row in the result set exposed through the ResultSet object) advances one row.
   - A boolean value returns to indicate whether there is a row to advance to.

5. **For each row, retrieve the values**

   Values are retrieved for each column in the ResultSet object by identifying either the name or position of the column. You can use the getData method to get the value from a column on the current row.

Java objects can use JDBC objects to interact with a database and get data for their own use.

Differences between client- and server-side JDBC connections

A difference between JDBC on the client and in the database server lies in establishing a connection with the database environment.

- **Client side** In client-side JDBC, establishing a connection requires the iAnywhere JDBC driver or the jConnect JDBC driver. Passing arguments to DriverManager.getConnection establishes the connection. The database environment is an external application from the perspective of the client application.
- **Server-side** When using JDBC within the database server, a connection already exists. The string "jdbc:default:connection" is passed to DriverManager.getConnection, which allows the JDBC application to work within the current user connection. This is a quick, efficient, and safe operation because the client application has already passed the database security to establish the connection. The user ID and password, having been provided once, do not need to be provided again. The server-side JDBC driver can only connect to the database of the current connection.

You can write JDBC classes so that they can run both at the client and at the server by employing a single conditional statement for constructing the URL. An external connection requires the host name and port number, while the internal connection requires "jdbc:default:connection".

Using the iAnywhere JDBC driver

The iAnywhere JDBC driver provides a JDBC driver that has some performance benefits and feature benefits compared to the pure Java jConnect JDBC driver, but which is not a pure-Java solution. The iAnywhere JDBC driver is recommended in most cases.

For information about choosing which JDBC driver to use, see "Choosing a JDBC driver" on page 478.

Loading the iAnywhere JDBC driver

Before you can use the iAnywhere JDBC driver in your application, you must load the appropriate driver. Load the iAnywhere JDBC 3.0 driver with the following statement:

```
DriverManager.registerDriver( (Driver)
    Class.forName(
    "ianywhere.ml.jdbcodbc.jdbc3.IDriver").newInstance()
    );
```

Using the newInstance method works around issues in some browsers.

- As the classes are loaded using Class.forName, the package containing the iAnywhere JDBC driver does not have to be imported using import statements.
- *jodbc.jar* must be in your classpath when you run the application.

  ```
  set classpath=%classpath%;install-dir\java\jodbc.jar
  ```

Required files

The Java component of the iAnywhere JDBC driver is included in the *jodbc.jar* file installed into the *Java* subdirectory of your SQL Anywhere installation. For Windows, the native component is *dbjodbc11.dll* in the *bin32* or *bin64* subdirectory of your SQL Anywhere installation; for Unix, the native component is *libdbjodbc11.so*. This component must be in the system path. When deploying applications using this driver, you must also deploy the ODBC driver files.

Supplying a URL to the driver

To connect to a database via the iAnywhere JDBC driver, you need to supply a URL for the database. For example:

```
Connection con = DriverManager.getConnection(
      "jdbc:ianywhere:DSN=SQL Anywhere 11 Demo" );
```

The URL contains **jdbc:ianywhere:** followed by a standard ODBC connection string. The connection string is commonly an ODBC data source, but you can also use explicit individual connection parameters, separated by semicolons, in addition to or instead of the data source. For more information about the parameters that you can use in a connection string, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

If you do not use a data source, you should specify the ODBC driver to use by including the DRIVER parameter in your connection string:

```
Connection con = DriverManager.getConnection(
      "jdbc:ianywhere:driver=SQL Anywhere 11;..." );
```


Using the jConnect JDBC driver

SQL Anywhere supports one version of jConnect: jConnect 6.0.5. The jConnect driver is available as a separate download from jConnect for JDBC. Documentation for jConnect can also be found on the same page.

If you want to use JDBC from an applet, you must use the jConnect JDBC driver to connect to SQL Anywhere databases.

The jConnect driver files

SQL Anywhere supports the following version of jConnect:

- **jConnect 6.0.5** This version of jConnect is for developing JDK 1.4 or later applications. jConnect 6.0.5 is JDBC 3.0 compliant. jConnect 6.0.5 is supplied as a JAR file named *jconn3.jar*.

> **Note**
> For the purposes of this documentation, all of the explanations and code samples provided assume that you are developing JDK 1.5 applications, and that you are using the jConnect 6.0.5 driver.
>
> There is a copy of *jconn3.jar* in the *install-dir\java* folder. However, we recommend that you use the version of the file included with jConnect 6.0.5 since it will be current with the version of jConnect that you have installed.

Setting the class path for jConnect

For your application to use jConnect, the jConnect classes must be in your class path at compile time and run time, so that the Java compiler and Java runtime can locate the necessary files.

The following command adds the jConnect 6.0.5 driver to an existing CLASSPATH environment variable (where *path* is your jConnect installation directory).

```
set classpath=path\jConnect-6_0\classes\jconn3.jar;%classpath%
```

Importing the jConnect classes

The classes in jConnect 6.0.5 are all in com.sybase.jdbc3.jdbc. You must import these classes at the beginning of each source file:

```
import com.sybase.jdbc3.jdbc.*
```

Encrypting passwords

SQL Anywhere supports password encryption for jConnect connections.

Installing jConnect system objects into a database

If you want to use jConnect to access system table information (database metadata), you must add the jConnect system objects to your database.

You can add the jConnect system objects to the database when creating the database or at a later time by upgrading the database. You can upgrade a database from Sybase Central or by using the dbupgrad utility.

Windows Mobile
Do not add jConnect system objects to your Windows Mobile database. See "Using jConnect on Windows Mobile" [*SQL Anywhere Server - Database Administration*].

Caution
You should always back up your database files before upgrading. If you apply the upgrade to the existing files, then these files become unusable if the upgrade fails. For information about backing up your database, see "Backup and data recovery" [*SQL Anywhere Server - Database Administration*].

To add jConnect system objects to a database (Sybase Central)

1. Connect to the database from Sybase Central as a DBA user.
2. Select the database, if necessary, and then from the **Tools** menu choose SQL Anywhere 11 and then **Upgrade Database**.

   The **Upgrade Database Wizard** appears.
3. Follow the instructions in the wizard to add jConnect support to the database.

To add jConnect system objects to a database (dbupgrad)

- From a command prompt, execute the following command:

  ```
  dbupgrad -c "connection-string"
  ```

  In this command, *connection-string* is a suitable connection string that enables access to a database and server as a DBA user.

Loading the jConnect driver

Before you can use jConnect in your application, load the driver with the following statement:

```
DriverManager.registerDriver( (Driver)
    Class.forName(
    "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
    );
```

Using the newInstance method works around issues in some browsers.

- As the classes are loaded using Class.forName, the package containing the jConnect driver does not have to be imported using import statements.
- To use jConnect 6.0.5, *jconn3.jar* must be in your classpath when you run the application. *jconn3.jar* is located in the *classes* subdirectory of your jConnect 6.0.5 installation (typically, *jConnect-6_0\classes*).

Supplying a URL to the driver

To connect to a database via jConnect, you need to supply a URL for the database. For example:

```
Connection con = DriverManager.getConnection(
      "jdbc:sybase:Tds:localhost:2638", "DBA", "sql");
```

The URL is composed in the following way:

```
jdbc:sybase:Tds:host:port
```

The individual components are:

- **jdbc:sybase:Tds** The jConnect JDBC driver, using the TDS application protocol.
- **host** The IP address or name of the computer on which the server is running. If you are establishing a same-host connection, you can use localhost, which means the computer system you are logged into.
- **port** The port number on which the database server listens. The port number assigned to SQL Anywhere is 2638. Use that number unless there are specific reasons not to do so.

The connection string must be less than 253 characters in length.

Specifying a database on a server

Each SQL Anywhere database server can have one or more databases loaded at a time. If the URL you supply when connecting via jConnect specifies a server, but does not specify a database, then the connection attempt is made to the default database on the server.

You can specify a particular database by providing an extended form of the URL in one of the following ways.

Using the ServiceName parameter

```
jdbc:sybase:Tds:host:port?ServiceName=database
```

The question mark followed by a series of assignments is a standard way of providing arguments to a URL. The case of ServiceName is not significant, and there must be no spaces around the = sign. The *database* parameter is the database name, not the server name. The database name must not include the path or file suffix. For example:

```
Connection con = DriverManager.getConnection(
      "jdbc:sybase:Tds:localhost:2638?ServiceName=demo", "DBA", "sql");
```

Using the RemotePWD parameter

A workaround exists for passing additional connection parameters to the server.

This technique allows you to provide additional connection parameters such as the database name, or a database file, using the RemotePWD field. You set RemotePWD as a Properties field using the put method.

The following code illustrates how to use the field.

```
import java.util.Properties;
.
```

```
.
.
DriverManager.registerDriver( (Driver)
    Class.forName(
    "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
    );

Properties props = new Properties();
props.put( "User", "DBA" );
props.put( "Password", "sql" );
props.put( "RemotePWD", ",DatabaseFile=mydb.db" );

Connection con = DriverManager.getConnection(
    "jdbc:sybase:Tds:localhost:2638", props );
```

As shown in the example, a comma must precede the DatabaseFile connection parameter. Using the DatabaseFile parameter, you can start a database on a server using jConnect. By default, the database is started with autostop=YES. If you specify utility_db with a DatabaseFile (DBF) or DatabaseName (DBN) connection parameter (for example, DBN=utility_db), then the utility database is started automatically.

For more information about the utility database, see "Using the utility database" [*SQL Anywhere Server - Database Administration*].

Database options set for jConnect connections

When an application connects to the database using the jConnect driver, the sp_tsql_environment stored procedure is called. The sp_tsql_environment procedure sets some database options for compatibility with Adaptive Server Enterprise behavior.

See also

- "Characteristics of Open Client and jConnect connections" [*SQL Anywhere Server - Database Administration*]
- "sp_tsql_environment system procedure" [*SQL Anywhere Server - SQL Reference*]

Connecting from a JDBC client application

Database metadata is always available when using the iAnywhere JDBC driver.

If you want to access database system tables (database metadata) from a JDBC application that uses jConnect, you must add a set of jConnect system objects to your database. These procedures are installed to all databases by default. The dbinit -i option prevents this installation.

For more information about adding the jConnect system objects to a database, see "Using the jConnect JDBC driver" on page 483.

The following complete Java application is a command line program that connects to a running database, prints a set of information to your command line, and terminates.

Establishing a connection is the first step any JDBC application must take when working with database data.

This example illustrates an external connection, which is a regular client/server connection. For information about how to create an internal connection from Java classes running inside the database server, see "Establishing a connection from a server-side JDBC class" on page 489.

Connection example code

The following is the source code for the methods used to make a connection. The source code can be found in the file *JDBCConnect.java* in the *samples-dir\SQLAnywhere\JDBC* directory. As presented, the example uses the JDBC 3.0 version of the iAnywhere JDBC driver to connect to the database. To use the jConnect 6.0.5 driver replace `ianywhere.ml.jdbcodbc.jdbc3.IDriver` with `com.sybase.jdbc3.jdbc.SybDriver`. You must also change the connection string if you want to use the jConnect 6.0.5 driver. Code alternatives are included as comments in the source code.

```
import java.io.*;
import java.sql.*;

public class JDBCConnect
{
  public static void main( String args[] )
  {
    try
    {
      // Select the JDBC driver. May throw a SQLException.
      // Choices are jConnect 6.0 driver
      // or iAnywhere JDBC 3.0 driver.
      // Currently, we use the iAnywhere JDBC 3.0 driver.
      DriverManager.registerDriver( (Driver)
        Class.forName(
        // "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
        "ianywhere.ml.jdbcodbc.jdbc3.IDriver").newInstance()
        );

      // Create a connection. Choices are TDS using jConnect,
      // Sun's JDBC-ODBC bridge, or the iAnywhere JDBC driver.
      // Currently, we use the iAnywhere JDBC driver.
      Connection con = DriverManager.getConnection(
        // "jdbc:sybase:Tds:localhost:2638", "DBA", "sql");
        // "jdbc:odbc:driver=SQL Anywhere 11;uid=DBA;pwd=sql" );
        "jdbc:ianywhere:driver=SQL Anywhere 11;uid=DBA;pwd=sql" );
```

```
            // Create a statement object, the container for the SQL
            // statement. May throw a SQLException.
            Statement stmt = con.createStatement();

            // Create a result set object by executing the query.
            // May throw a SQLException.
            ResultSet rs = stmt.executeQuery(
              "SELECT ID, GivenName, Surname FROM Customers");

            // Process the result set.
            while (rs.next())
            {
              int value = rs.getInt(1);
              String FirstName = rs.getString(2);
              String LastName = rs.getString(3);
              System.out.println(value+" "+FirstName+" "+LastName);
            }
            rs.close();
            stmt.close();
            con.close();
          }
          catch (SQLException sqe)
          {
            System.out.println("Unexpected exception : " +
                       sqe.toString() + ", sqlstate = " +
                       sqe.getSQLState());
            System.exit(1);
          }
          catch (Exception e)
          {
            e.printStackTrace();
            System.exit(1);
          }

          System.exit(0);
        }
      }
```

How the connection example works

The external connection example is a Java command line program.

Importing packages

The application requires a couple of packages, which are imported in the first lines of *JDBCConnect.java*:

- The java.io package contains the Sun Microsystems io classes, which are required for printing to the console window.
- The java.sql package contains the Sun Microsystems JDBC classes, which are required for all JDBC applications.

The main method

Each Java application requires a class with a method named main, which is the method invoked when the program starts. In this simple example, JDBCConnect.main is the only public method in the application.

The JDBCConnect.main method carries out the following tasks:

1. Loads the iAnywhere JDBC 3.0 driver (identified by the driver string "ianywhere.ml.jdbcodbc.jdbc3.IDriver").

   Class.forName loads the driver. Using the newInstance method works around issues in some browsers.

2. Connects to the default running database using an iAnywhere JDBC driver URL. If you are using the jConnect driver instead, use the URL "jdbc:sybase:Tds:localhost:2638" (as shown in comments) with "DBA" and "sql" as the user ID and password, respectively.

   DriverManager.getConnection establishes a connection using the specified URL.

3. Creates a statement object, which is the container for the SQL statement.
4. Creates a result set object by executing a SQL query.
5. Iterates through the result set, printing the column information.
6. Closes each of the result set, statement, and connection objects.

Running the connection example

To create and execute the external connection example application

1. At a command prompt, change to the *samples-dir\SQLAnywhere\JDBC* directory.
2. Start a database server with the sample database on your local computer using the following command:

   ```
   dbeng11 samples-dir\demo.db
   ```

3. Set the CLASSPATH environment variable.

   ```
   set classpath=%classpath%;install-dir\java\jodbc.jar
   ```

   If you are using the jConnect driver instead, then use the following (where *path* is your jConnect installation directory):

   ```
   set classpath=path\jConnect-6_0\classes\jconn3.jar;%classpath%
   ```

4. Run the following command to compile the example:

   ```
   javac JDBCConnect.java
   ```

5. Run the following command to run the example:

   ```
   java JDBCConnect
   ```

6. Confirm that a list of identification numbers with customer's names appears at the command prompt.

   If the attempt to connect fails, an error message appears instead. Confirm that you have executed all the steps as required. Check that your class path is correct. An incorrect setting may result in a failure to locate a class.

Establishing a connection from a server-side JDBC class

SQL statements in JDBC are built using the createStatement method of a Connection object. Even classes running inside the server need to establish a connection to create a Connection object.

Establishing a connection from a server-side JDBC class is more straightforward than establishing an external connection. Because the user is already connected to the database, the class simply uses the current connection.

Server-side connection example code

The following is the source code for the server-side connection example. It is a modified version of the source code in *samples-dir\SQLAnywhere\JDBC\JDBCConnect.java*.

```
import java.io.*;
import java.sql.*;

public class JDBCConnect2
{
  public static void main( String args[] )
  {
    try
    {
      // Open the connection. May throw a SQLException.
      Connection con = DriverManager.getConnection(
          "jdbc:default:connection" );

      // Create a statement object, the container for the SQL
      // statement. May throw a SQLException.
      Statement stmt = con.createStatement();
      // Create a result set object by executing the query.
      // May throw a SQLException.
      ResultSet rs = stmt.executeQuery(
        "SELECT ID, GivenName, Surname FROM Customers");

      // Process the result set.
      while (rs.next())
      {
        int value = rs.getInt(1);
        String FirstName = rs.getString(2);
        String LastName = rs.getString(3);
        System.out.println(value+" "+FirstName+" "+LastName);
      }
      rs.close();
      stmt.close();
      con.close();
    }
    catch (SQLException sqe)
    {
      System.out.println("Unexpected exception : " +
                sqe.toString() + ", sqlstate = " +
                sqe.getSQLState());
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```


How the server-side connection example differs

The server-side connection example is almost identical to the client-side connection example, with the following exceptions:

1. The driver manager does not need to be loaded. The following code has been removed from the example.

```
DriverManager.registerDriver( (Driver)
  Class.forName(
  // "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
  "ianywhere.ml.jdbcodbc.IDriver").newInstance()
  );
```

2. It connects to the default running database using the current connection. The URL in the getConnection call has been changed as follows:

```
Connection con = DriverManager.getConnection(
          "jdbc:default:connection" );
```

3. The `System.exit()` statements have been removed.

Running the server-side connection example

To create and execute the internal connection example application

1. At a command prompt, change to the *samples-dir\SQLAnywhere\JDBC* directory.
2. Start a database server with the sample database on your local computer using the following command:

```
dbeng11 samples-dir\demo.db
```

3. For server-side JDBC, it is not necessary to set the CLASSPATH environment variable.
4. Enter the following command to compile the example:

```
javac JDBCConnect2.java
```

5. Install the class into the sample database using Interactive SQL. Run the following statement:

```
INSTALL JAVA NEW
FROM FILE 'samples-dir\SQLAnywhere\JDBC\JDBCConnect2.class'
```

   You can also install the class using Sybase Central. While connected to the sample database, open the **Java** subfolder under **External Environments** and choose **File** » **New** » **Java Class**. Then follow the instructions in the wizard.

6. Define a stored procedure named JDBCConnect that acts as a wrapper for the JDBCConnect2.main method in the class:

```
CREATE PROCEDURE JDBCConnect()
  EXTERNAL NAME 'JDBCConnect2.main([Ljava/lang/String;)V'
  LANGUAGE JAVA;
```

7. Call the JDBCConnect2.main method as follows:

```
call JDBCConnect();
```


The first time a Java class is called in a session, the Java VM must be loaded. This might take a few seconds.

8. Confirm that a list of identification numbers with customers' names appears in the database server messages window.

   If the attempt to connect fails, an error message appears instead. Confirm that you have executed all the steps as required.

Notes on JDBC connections

- **Autocommit behavior** The JDBC specification requires that, by default, a COMMIT is performed after each data modification statement. Currently, the client-side JDBC behavior is to commit (autocommit is true) and the server-side behavior is to not commit (autocommit is false). To obtain the same behavior in both client-side and server-side applications, you can use a statement such as the following:

  ```
  con.setAutoCommit( false );
  ```

  In this statement, con is the current connection object. You could also set autocommit to true.

- **Connection defaults** From server-side JDBC, only the first call to `getConnection( "jdbc:default:connection" )` creates a new connection with the default values. Subsequent calls return a wrapper of the current connection with all connection properties unchanged. If you set autocommit to false in your initial connection, any subsequent getConnection calls within the same Java code return a connection with autocommit set to false.

  You may want to ensure that closing a connection restores the connection properties to their default values, so that subsequent connections are obtained with standard JDBC values. The following code achieves this:

  ```
  Connection con =
      DriverManager.getConnection("jdbc:default:connection");

  boolean oldAutoCommit = con.getAutoCommit();
  try {
       // main body of code here
  }
  finally {
      con.setAutoCommit( oldAutoCommit );
  }
  ```

  This discussion applies not only to autocommit, but also to other connection properties such as transaction isolation level and read-only mode.

  For more information about the getTransactionIsolation, setTransactionIsolation, and isReadOnly methods, see documentation on the java.sql.Connection interface.

Using JDBC to access data

Java applications that hold some or all classes in the database have significant advantages over traditional SQL stored procedures. At an introductory level, however, it may be helpful to use the parallels with SQL stored procedures to demonstrate the capabilities of JDBC. In the following examples, you write Java classes that insert a row into the Departments table.

As with other interfaces, SQL statements in JDBC can be either **static** or **dynamic**. Static SQL statements are constructed in the Java application and sent to the database. The database server parses the statement, selects an execution plan, and executes the statement. Together, parsing and selecting an execution plan are referred to as **preparing** the statement.

If a similar statement has to be executed many times (many inserts into one table, for example), there can be significant overhead in static SQL because the preparation step has to be executed each time.

In contrast, a dynamic SQL statement contains placeholders. The statement, prepared once using these placeholders, can be executed many times without the additional expense of preparing. Dynamic SQL is discussed in "Using prepared statements for more efficient access" on page 495.

Preparing for the examples

Sample code

The code fragments in this section are taken from the complete class in *samples-dir\SQLAnywhere\JDBC\JDBCExample.java*.

To install the JDBCExample class

1. Compile the *JDBCExample.java* source code.
2. Using Interactive SQL, connect to the sample database as the DBA.
3. Install the *JDBCExample.class* file into the sample database by executing the following statement in Interactive SQL (*samples-dir* represents the SQL Anywhere samples directory):

   ```
   INSTALL JAVA NEW
   FROM FILE 'samples-dir\SQLAnywhere\JDBC\JDBCExample.class'
   ```

   You can also install the class using Sybase Central. While connected to the sample database, open the **Java** subfolder under **External Environments** and choose **File** » **New** » **Java Class**. Follow the instructions in the wizard.

Inserts, updates, and deletes using JDBC

The Statement object executes static SQL statements. You execute SQL statements such as INSERT, UPDATE, and DELETE, which do not return result sets, using the executeUpdate method of the Statement object. Statements, such as CREATE TABLE and other data definition statements, can also be executed using executeUpdate.

The following code fragment illustrates how to execute an INSERT statement. It uses a Statement object that has been passed to the InsertStatic method as an argument.

```
public static void InsertStatic( Statement stmt )
{
  try
  {
    int iRows = stmt.executeUpdate(
      "INSERT INTO Departments (DepartmentID, DepartmentName)"
      + " VALUES (201, 'Eastern Sales')" );
    // Print the number of rows inserted
    System.out.println(iRows + " rows inserted");
  }
  catch (SQLException sqe)
  {
    System.out.println("Unexpected exception : " +
              sqe.toString() + ", sqlstate = " +
              sqe.getSQLState());
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

Source code available
This code fragment is part of the JDBCExample class included in the *samples-dir\SQLAnywhere\JDBC* directory.

Notes

- The executeUpdate method returns an integer that reflects the number of rows affected by the operation. In this case, a successful INSERT would return a value of one (1).
- When run as a server-side class, the output from `System.out.println` goes to the database server messages window.

To run the JDBC Insert example

1. Using Interactive SQL, connect to the sample database as the DBA.
2. Ensure the JDBCExample class has been installed.

   For more information about installing the Java examples classes, see "Preparing for the examples" on page 493.
3. Define a stored procedure named JDBCExample that acts as a wrapper for the JDBCExample.main method in the class:

   ```
   CREATE PROCEDURE JDBCExample( IN arg CHAR(50) )
     EXTERNAL NAME 'JDBCExample.main([Ljava/lang/String;)V'
     LANGUAGE JAVA;
   ```
4. Call the JDBCExample.main method as follows:

   ```
   CALL JDBCExample( 'insert' );
   ```


The argument string `'insert'` causes the InsertStatic method to be invoked.

5. Confirm that a row has been added to the Departments table.

   ```
   SELECT * FROM Departments;
   ```

   The example program displays the updated contents of the Departments table in the database server messages window.

6. There is a similar method in the example class called DeleteStatic that shows how to delete the row that has just been added. Call the JDBCExample.main method as follows:

   ```
   CALL JDBCExample( 'delete' );
   ```

   The argument string `'delete'` causes the DeleteStatic method to be invoked.

7. Confirm that the row has been deleted from the Departments table.

   ```
   SELECT * FROM Departments;
   ```

   The example program displays the updated contents of the Departments table in the database server messages window.

Using prepared statements for more efficient access

If you use the Statement interface, you parse each statement that you send to the database, generate an access plan, and execute the statement. The steps prior to actual execution are called **preparing** the statement.

You can achieve performance benefits if you use the PreparedStatement interface. This allows you to prepare a statement using placeholders, and then assign values to the placeholders when executing the statement.

Using prepared statements is particularly useful when carrying out many similar actions, such as inserting many rows.

For more information about prepared statements, see “Preparing statements” on page 24.

Example

The following example illustrates how to use the PreparedStatement interface, although inserting a single row is not a good use of prepared statements.

The following InsertDynamic method of the JDBCExample class carries out a prepared statement:

```
public static void InsertDynamic( Connection con,
                String ID, String name )
{
  try {
    // Build the INSERT statement
    // ? is a placeholder character
    String sqlStr = "INSERT INTO Departments " +
            "( DepartmentID, DepartmentName ) " +
            "VALUES ( ? , ? )";

    // Prepare the statement
    PreparedStatement stmt =
      con.prepareStatement( sqlStr );
```

```
      // Set some values
      int idValue = Integer.valueOf( ID );
      stmt.setInt( 1, idValue );
      stmt.setString( 2, name );

      // Execute the statement
      int iRows = stmt.executeUpdate();

      // Print the number of rows inserted
      System.out.println(iRows + " rows inserted");
    }
    catch (SQLException sqe)
    {
      System.out.println("Unexpected exception : " +
                 sqe.toString() + ", sqlstate = " +
                 sqe.getSQLState());
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
```

Source code available
This code fragment is part of the JDBCExample class included in the *samples-dir\SQLAnywhere\JDBC* directory.

Notes

- The executeUpdate method returns an integer that reflects the number of rows affected by the operation. In this case, a successful INSERT would return a value of one (1).
- When run as a server-side class, the output from `System.out.println` goes to the database server messages window.

To run the JDBC Insert example

1. Using Interactive SQL, connect to the sample database as the DBA.
2. Ensure the JDBCExample class has been installed.

   For more information about installing the Java examples classes, see "Preparing for the examples" on page 493.
3. Define a stored procedure named JDBCInsert that acts as a wrapper for the JDBCExample.Insert method in the class:

   ```
   CREATE PROCEDURE JDBCInsert( IN arg1 INTEGER, IN arg2 CHAR(50) )
     EXTERNAL NAME 'JDBCExample.Insert(ILjava/lang/String;)V'
     LANGUAGE JAVA;
   ```
4. Call the JDBCExample.Insert method as follows:

   ```
   CALL JDBCInsert( 202, 'Southeastern Sales' );
   ```

   The Insert method causes the InsertDynamic method to be invoked.
5. Confirm that a row has been added to the Departments table.


```
SELECT * FROM Departments;
```

The example program displays the updated contents of the Departments table in the database server messages window.

6. There is a similar method in the example class called DeleteDynamic that shows how to delete the row that has just been added.

   Define a stored procedure named JDBCDelete that acts as a wrapper for the JDBCExample.Delete method in the class:

   ```
   CREATE PROCEDURE JDBCDelete( in arg1 integer )
     EXTERNAL NAME 'JDBCExample.Delete(I)V'
     LANGUAGE JAVA;
   ```

7. Call the JDBCExample.Delete method as follows:

   ```
   CALL JDBCDelete( 202 );
   ```

   The Delete method causes the DeleteDynamic method to be invoked.

8. Confirm that the row has been deleted from the Departments table.

   ```
   SELECT * FROM Departments;
   ```

   The example program displays the updated contents of the Departments table in the database server messages window.

Using prepared statements for wide inserts

The PreparedStatement.addBatch() method is useful for performing batched (or wide) inserts. The following are some guidelines to using this method.

1. An INSERT statement should be prepared using one of the Connection.prepareStatement() methods.

   ```
   // Build the INSERT statement
   String sqlStr = "INSERT INTO Departments " +
               "( DepartmentID, DepartmentName ) " +
               "VALUES ( ? , ? )";
   // Prepare the statement
   PreparedStatement stmt =
        con.prepareStatement( sqlStr );
   ```

2. The parameters for the prepared insert statement should be set and batched as follows:

   ```
   // loop to batch "n" sets of parameters
   for( i=0; i < n; i++ )
   {
       // Note "stmt" is the original prepared insert statement from step 1.
       stmt.setSomeType( 1, param_1 );
       stmt.setSomeType( 2, param_2 );
       .
       .
       .
       // Note that there are "m" parameters in the statement.
       stmt.setSomeType( m , param_m );

       // Add the set of parameters to the batch and
   ```

```
        // move to the next row of parameters.
        stmt.addBatch();
    }
```

Example:

```
    for( i=0; i < 5; i++ )
    {
        stmt.setInt( 1, idValue );
        stmt.setString( 2, name );
        stmt.addBatch();
    }
```

3. The batch should then be executed using the PreparedStatement.executeUpdate() method.

It should be noted that only the PreparedStatement.addBatch() method is supported and that the PreparedStatement.executeUpdate() method needs to be called to execute the batch. None of batch methods for the Statement object (i.e. Statement.addBatch(), Statement.clearBatch(), Statement.executeBatch()) are supported since these methods are completely optional and not very useful. For such static batches, it is best to call Statement.execute() or Statement.executeQuery() on a single string with the batched statements wrapped inside a BEGIN...END.

Notes

- BLOB parameters are not supported in batches.
- String/Binary parameters are supported but the size of the string/binary parameter is an issue. By default the string/binary parameter is restricted to 255 characters or 510 bytes. The reason for the restriction is due to the underlying ODBC protocol and will not be discussed here. For further information, it is best to view the documentation on passing arrays of parameters in ODBC. If, however, an application needs to pass larger string or binary parameters within a batch then an additional method, setBatchStringSize, has been provided to increase the size of the string/binary parameter. Note that this method must be called prior to the first addBatch() call. If the method is called after the first addBatch() call, then the new size setting will be ignored. Hence, when calling this method to change the size of a string/binary parameter, the application needs to know ahead of time what the maximum string or binary value for that parameter will be.

To use the setBatchStringSize method, you must modify the "code" above as follows:

```
// You need to cast "stmt" to an IPreparedStatment object
// in order to change the size of string/binary parameters.
ianywhere.ml.jdbcodbc.IPreparedStatement _stmt =
    (ianywhere.ml.jdbcodbc.IPreparedStatement)stmt;

// Now, for example, change the size of string parameter 4
// from the default 255 characters to 300 characters.
// Note that string parameters are measured in "characters".
_stmt.setBatchStringSize( 4, 300 );

// Change the size of binary parameter 6
// from the default 510 bytes to 750 bytes.
// Note that binary parameters are measured in "bytes".
_stmt.setBatchStringSize( 6, 750 );

// loop to batch "n" sets of parameters
// where n should not be too large
for( i=0; i < n; i++ )
{
```



```
        // stmt is the prepared insert statement from step 1
        stmt.setSomeType( 1, param_1 );
        stmt.setSomeType( 2, param_2 );
        .
        .
        .
        // Note that there are "m" parameters in the statement.
        stmt.setSomeType( m , param_m );

        // Add the set of parameters to the batch and
        // move to the next row of parameters.
        stmt.addBatch();
    }
```

The maximum string/binary size of a parameter should be modified with caution. If the maximum is set too high, then the additional memory allocation cost will probably override any performance gained from using the batch. Also, there is a good chance the application will not know ahead of time what the maximum string or binary value for a particular parameter is. The recommendation then is to not change the maximum string or binary size of a parameter, but rather to use the batch method until a string or binary value larger than the current/default maximum is encountered. The application can, at that point, call executeBatch() to execute the parameters that are currently batched, then call the regular set and executeUpdate() method to execute until the large string/binary parameters have been handled, then switch back into batch mode when smaller string/binary parameters are encountered.

Returning result sets

This section describes how to make one or more result sets available from Java methods.

You must write a Java method that returns one or more result sets to the calling environment, and wrap this method in a SQL stored procedure. The following code fragment illustrates how multiple result sets can be returned to the calling SQL code. It uses three executeQuery statements to obtain three different result sets.

```
public static void Results( ResultSet[] rset )
       throws SQLException
{
    // Demonstrate returning multiple result sets

    Connection con = DriverManager.getConnection(
                    "jdbc:default:connection" );
    rset[0] = con.createStatement().executeQuery(
        "SELECT * FROM Employees" +
        "   ORDER BY EmployeeID" );
    rset[1] = con.createStatement().executeQuery(
        "SELECT * FROM Departments" +
        "   ORDER BY DepartmentID" );
    rset[2] = con.createStatement().executeQuery(
        "SELECT i.ID,i.LineID,i.ProductID,i.Quantity," +
        "       s.OrderDate,i.ShipDate," +
        "       s.Region,e.GivenName||' '||e.Surname" +
        "   FROM SalesOrderItems AS i" +
        "   JOIN SalesOrders AS s" +
        "   JOIN Employees AS e" +
        "   WHERE s.ID=i.ID" +
        "         AND s.SalesRepresentative=e.EmployeeID" );
    con.close();
}
```

Source code available
This code fragment is part of the JDBCExample class included in the *samples-dir\SQLAnywhere\JDBC* directory.

Notes

- This server-side JDBC example connects to the default running database using the current connection using getConnection.
- The executeQuery methods return result sets.

To run the JDBC result set example

1. Using Interactive SQL, connect to the sample database as the DBA.
2. Ensure the JDBCExample class has been installed.

   For more information about installing the Java examples classes, see "Preparing for the examples" on page 493.
3. Define a stored procedure named JDBCResults that acts as a wrapper for the JDBCExample.Results method in the class:

   ```
   CREATE PROCEDURE JDBCResults()
     DYNAMIC RESULT SETS 3
     EXTERNAL NAME 'JDBCExample.Results([Ljava/sql/ResultSet;)V'
     LANGUAGE JAVA;
   ```
4. Set the following Interactive SQL options so you can see all the results of the query:
   a. From the **Tools** menu, choose **Options**.

      The **Options** window appears.
   b. Click **SQL Anywhere**.
   c. Click the **Results** tab.
   d. Set the value for **Maximum Number Of Rows To Display** to **5000**.
   e. Select **Show All Result Sets**.
   f. Click **OK**.
5. Call the JDBCExample.Results method as follows:

   ```
   CALL JDBCResults();
   ```
6. Check each of the three results tabs, Result Set 1, Result Set 2, and Result Set 3.

Miscellaneous JDBC notes

- **Access permissions** Like all Java classes in the database, classes containing JDBC statements can be accessed by any user provided that the GRANT EXECUTE statement has granted them permission to execute the stored procedure that is acting as a wrapper for the Java method.

- **Execution permissions** Java classes are executed with the permissions of the connection executing them. This behavior is different from that of stored procedures, which execute with the permissions of the owner.

Using JDBC escape syntax

You can use JDBC escape syntax from any JDBC application, including Interactive SQL. This escape syntax allows you to call stored procedures regardless of the database management system you are using. The general form for the escape syntax is

{{ **keyword** *parameters* }}

In Interactive SQL, the braces *must* be doubled. There must not be a space between successive braces: "{{" is acceptable, but "{ {" is not. As well, you cannot use newline characters in the statement. The escape syntax cannot be used in stored procedures because they are not executed by Interactive SQL.

You can use the escape syntax to access a library of functions implemented by the JDBC driver that includes number, string, time, date, and system functions.

For example, to obtain the name of the current user in a database management system-neutral way, you would execute the following:

```
SELECT {{ FN USER() }}
```

The functions that are available depend on the JDBC driver that you are using. The following tables list the functions that are supported by the iAnywhere JDBC driver and by the jConnect driver.

iAnywhere JDBC driver supported functions

Numeric functions	String functions	System functions	Time/date functions
ABS	ASCII	IFNULL	CURDATE
ACOS	CHAR	USERNAME	CURTIME
ASIN	CONCAT		DAYNAME
ATAN	DIFFERENCE		DAYOFMONTH
ATAN2	INSERT		DAYOFWEEK
CEILING	LCASE		DAYOFYEAR
COS	LEFT		HOUR
COT	LENGTH		MINUTE
DEGREES	LOCATE		MONTH
EXP	LOCATE_2		MONTHNAME
FLOOR	LTRIM		NOW
LOG	REPEAT		QUARTER

Numeric functions	String functions	System functions	Time/date functions
LOG10	RIGHT		SECOND
MOD	RTRIM		WEEK
PI	SOUNDEX		YEAR
POWER	SPACE		
RADIANS	SUBSTRING		
RAND	UCASE		
ROUND			
SIGN			
SIN			
SQRT			
TAN			
TRUNCATE			

jConnect supported functions

Numeric functions	String functions	System functions	Time/date functions
ABS	ASCII	DATABASE	CURDATE
ACOS	CHAR	IFNULL	CURTIME
ASIN	CONCAT	USER	DAYNAME
ATAN	DIFFERENCE	CONVERT	DAYOFMONTH
ATAN2	LCASE		DAYOFWEEK
CEILING	LENGTH		HOUR
COS	REPEAT		MINUTE
COT	RIGHT		MONTH
DEGREES	SOUNDEX		MONTHNAME
EXP	SPACE		NOW

Numeric functions	String functions	System functions	Time/date functions
FLOOR	SUBSTRING		QUARTER
LOG	UCASE		SECOND
LOG10			TIMESTAMPADD
PI			TIMESTAMPDIFF
POWER			YEAR
RADIANS			
RAND			
ROUND			
SIGN			
SIN			
SQRT			
TAN			

A statement using the escape syntax should work in SQL Anywhere, Adaptive Server Enterprise, Oracle, SQL Server, or another database management system to which you are connected.

For example, to obtain database properties with the sa_db_info procedure using SQL escape syntax, you would execute the following in Interactive SQL:

```
{{CALL sa_db_info( 0 ) }}
```


iAnywhere JDBC 3.0 API support

All mandatory classes and methods of the JDBC 3.0 specification are supported. Some optional methods of the java.sql.Blob interface are not supported. These optional methods are:

- long position(Blob pattern, long start);
- long position(byte[] pattern, long start);
- OutputStream setBinaryStream(long pos)
- int setBytes(long pos, byte[] bytes)
- int setBytes(long pos, byte[] bytes, int offset, int len);
- void truncate(long len);

CHAPTER 14

SQL Anywhere embedded SQL

Contents

Introduction to embedded SQL

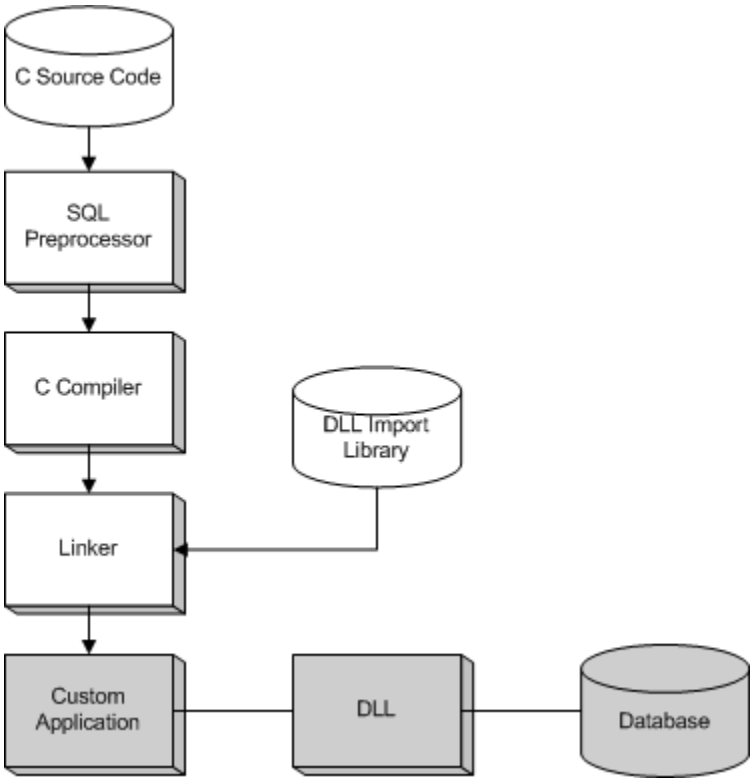
Embedded SQL is a database programming interface for the C and C++ programming languages. It consists of SQL statements intermixed with (embedded in) C or C++ source code. These SQL statements are translated by a **SQL preprocessor** into C or C++ source code, which you then compile.

At runtime, embedded SQL applications use a SQL Anywhere **interface library** to communicate with a database server. The interface library is a dynamic link library (**DLL**) or shared library on most platforms.

- On Windows operating systems, the interface library is *dblib11.dll*.
- On Unix operating systems, the interface library is *libdblib11.so*, *libdblib11.sl*, or *libdblib11.a*, depending on the operating system.
- On Mac OS X, the interface library is *libdblib11.dylib.1*.

SQL Anywhere provides two flavors of embedded SQL. Static embedded SQL is simpler to use, but is less flexible than dynamic embedded SQL.

Development process overview



Once the program has been successfully preprocessed and compiled, it is linked with the **import library** for the SQL Anywhere interface library to form an executable file. When the database server is running, this executable file uses the SQL Anywhere DLL to interact with the database server. The database server does not have to be running when the program is preprocessed.

For Windows, there are separate import libraries for Watcom C/C++, for Microsoft Visual C++, and for Borland C++.

Using import libraries is the standard development method for applications that call functions in DLLs. SQL Anywhere also provides an alternative, and recommended method which avoids the use of import libraries. For more information, see "Loading the interface library dynamically" on page 512.

Running the SQL preprocessor

The SQL preprocessor is an executable named *sqlpp.exe*.

The SQLPP command line is as follows:

sqlpp [*options*] *sql-filename* [*output-filename*]

The SQL preprocessor processes a C program with embedded SQL before the C or C++ compiler is run. The preprocessor translates the SQL statements into C/C++ language source that is put into the output file. The normal extension for source programs with embedded SQL is *.sqc*. The default output file name is the *sql-filename* with an extension of *.c*. If the *sql-filename* already has a *.c* extension, then the output file name extension is *.cc* by default.

Reprocessing embedded SQL
When an application is rebuilt to use a new major version of the database interface library, the embedded SQL files must be preprocessed with the same version's SQL preprocessor.

For a full listing of the command line options, see "SQL preprocessor" on page 566.

Supported compilers

The C language SQL preprocessor has been used in conjunction with the following compilers:

Operating system	Compiler	Version
Windows	Watcom C/C++	9.5 or later
Windows	Microsoft Visual C++	6.0 or later
Windows	Borland C++	4.5
Windows Mobile	Microsoft Visual C++	2005

Operating system	Compiler	Version
Windows Mobile	Microsoft eMbedded Visual C++	3.0, 4.0
Unix	GNU or native compiler	

Embedded SQL header files

All header files are installed in the *SDK\Include* subdirectory of your SQL Anywhere installation directory.

File name	Description
sqlca.h	Main header file included in all embedded SQL programs. This file includes the structure definition for the SQL Communication Area (SQLCA) and prototypes for all embedded SQL database interface functions.
sqlda.h	SQL Descriptor Area structure definition included in embedded SQL programs that use dynamic SQL.
sqldef.h	Definition of embedded SQL interface data types. This file also contains structure definitions and return codes needed for starting the database server from a C program.
sqlerr.h	Definitions for error codes returned in the sqlcode field of the SQLCA.
sqlstate.h	Definitions for ANSI/ISO SQL standard error states returned in the sqlstate field of the SQLCA.
pshpk1.h, pshpk2.h, poppk.h	These headers ensure that structure packing is handled correctly.

Import libraries

All import libraries are installed in the *SDK\Lib* subdirectories, under the SQL Anywhere installation directory. For example, Windows and Unix import libraries are stored in the *SDK\Lib\x86* and *SDK\Lib\x64* subdirectories. Windows Mobile import libraries are installed in the *SDK\Lib\CE\Arm.50* subdirectory.

Operating system	Compiler	Import library
Windows	Watcom C/C++ (32-bit only)	*dblibtw.lib*
Windows	Microsoft Visual C++	*dblibtm.lib*
Windows	Borland Delphi (32-bit only)	*dblibtb.lib*

Operating system	Compiler	Import library
Windows Mobile	Microsoft Visual C++ 2005	*dblib11.lib*
Windows Mobile	Microsoft eMbedded Visual C++	*dblib11.lib*
Solaris (unthreaded applications)	All compilers	*libdblib11.so*, *libdbtasks11.so*
Solaris (threaded applications)	All compilers	*libdblib11_r.so*, *libdbtasks11_r.so*

The *libdbtasks11* libraries are called by the *libdblib11* library. Some compilers locate *libdbtasks11* automatically, while for others you need to specify it explicitly.

A simple example

The following is a very simple example of an embedded SQL program.

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main()
{
 db_init( &sqlca );
 EXEC SQL WHENEVER SQLERROR GOTO error;
 EXEC SQL CONNECT "DBA" IDENTIFIED BY "sql";
 EXEC SQL UPDATE Employees
  SET Surname =  'Plankton'
  WHERE EmployeeID = 195;
 EXEC SQL COMMIT WORK;
 EXEC SQL DISCONNECT;
 db_fini( &sqlca );
 return( 0 );
error:
 printf( "update unsuccessful -- sqlcode = %ld\n",
  sqlca.sqlcode );
 db_fini( &sqlca );
 return( -1 );
}
```

This example connects to the database, updates the last name of employee number 195, commits the change, and exits. There is virtually no interaction between the SQL and C code. The only thing the C code is used for in this example is control flow. The WHENEVER statement is used for error checking. The error action (GOTO in this example) is executed after any SQL statement that causes an error.

For a description of fetching data, see "Fetching data" on page 550.

Structure of embedded SQL programs

SQL statements are placed (embedded) within regular C or C++ code. All embedded SQL statements start with the words EXEC SQL and end with a semicolon (;). Normal C language comments are allowed in the middle of embedded SQL statements.

Every C program using embedded SQL must contain the following statement before any other embedded SQL statements in the source file.

```
EXEC SQL INCLUDE SQLCA;
```

The first embedded SQL statement executed by the C program must be a CONNECT statement. The CONNECT statement is used to establish a connection with the database server and to specify the user ID that is used for authorizing all statements executed during the connection.

Some embedded SQL statements do not generate any C code, or do not involve communication with the database. These statements are allowed before the CONNECT statement. Most notable are the INCLUDE statement and the WHENEVER statement for specifying error processing.

Loading the interface library dynamically

The usual practice for developing applications that use functions from DLLs is to link the application against an **import library**, which contains the required function definitions.

This section describes an alternative to using an import library for developing SQL Anywhere applications. The SQL Anywhere interface library can be loaded dynamically, without having to link against the import library, using the *esqldll.c* module in the *SDK\C* subdirectory of your installation directory.

To load the interface DLL dynamically

1. Your program must call db_init_dll to load the DLL, and must call db_fini_dll to free the DLL. The db_init_dll call must be before any function in the database interface, and no function in the interface can be called after db_fini_dll.

   You must still call the db_init and db_fini library functions.

2. You must #include the *esqldll.h* header file before the EXEC SQL INCLUDE SQLCA statement or #include <*sqlca.h*> line in your embedded SQL program.

3. A SQL OS macro must be defined. The header file *sqlca.h*, which is included by *esqdll.c*, attempts to determine the appropriate macro and define it. However, certain combinations of platforms and compilers may cause this to fail. In this case, you must add a #define to the top of this file, or make the definition using a compiler option.

Macro	Platforms
_SQL_OS_UNIX	Unix
_SQL_OS_UNIX64	64-bit Unix

Macro	Platforms
_SQL_OS_WINDOWS	All Windows operating systems

4. Compile *esqldll.c*.
5. Instead of linking against the imports library, link the object module *esqldll.obj* with your embedded SQL application objects.

Sample

You can find a sample program illustrating how to load the interface library dynamically in the *samples-dir\SQLAnywhere\ESQLDynamicLoad* directory. The source code is in *sample.sqc*.

Sample embedded SQL programs

Sample embedded SQL programs are included with the SQL Anywhere installation. They are placed in the *samples-dir\SQLAnywhere\C* directory. For Windows Mobile, an additional example is located in the *samples-dir\SQLAnywhere\CE\esql_sample* directory.

- The static cursor embedded SQL example, *cur.sqc*, demonstrates the use of static SQL statements.
- The dynamic cursor embedded SQL example, *dcur.sqc*, demonstrates the use of dynamic SQL statements.

To reduce the amount of code that is duplicated by the sample programs, the mainlines and the data printing functions have been placed into a separate file. This is *mainch.c* for character mode systems and *mainwin.c* for windowing environments.

The sample programs each supply the following three routines, which are called from the mainlines:

- **WSQLEX_Init** Connects to the database and opens the cursor.
- **WSQLEX_Process_Command** Processes commands from the user, manipulating the cursor as necessary.
- **WSQLEX_Finish** Closes the cursor and disconnects from the database.

The function of the mainline is to:

1. Call the WSQLEX_Init routine.
2. Loop, getting commands from the user and calling WSQL_Process_Command until the user quits.
3. Call the WSQLEX_Finish routine.

Connecting to the database is accomplished with the embedded SQL CONNECT statement supplying the appropriate user ID and password.

In addition to these samples, you may find other programs and source files as part of SQL Anywhere that demonstrate features available for particular platforms.

Building the sample programs

Files to build the sample programs are supplied with the sample code.

- For Windows, use *build.bat* or *build64.bat* to compile the sample programs.

  For x64 platform builds, you may need to set up the correct environment for compiling and linking. Here is an example that builds the sample programs for an x64 platform.

  ```
  set mssdk=c:\MSSDK\v6.1
  build64
  ```

- For Unix, use the shell script *build.sh*.
- For Windows Mobile, use the *esql_sample.sln* project file for Microsoft Visual C++. This file appears in *samples-dir\SQLAnywhere\CE\esql_sample*.

This will build the following examples.

- **CUR** An embedded SQL static cursor example
- **DCUR** An embedded SQL dynamic cursor example
- **ODBC** An ODBC example which is discussing in "ODBC samples" on page 449.

Running the sample programs

The executable files and corresponding source code are located in the *samples-dir\SQLAnywhere\C* directory. For Windows Mobile, an additional example is located in the *samples-dir\SQLAnywhere\CE\esql_sample* directory.

To run the static cursor sample program

1. Start the SQL Anywhere sample database, *demo.db*.
2. For 32-bit Windows, run the file *curwin.exe*.

   For 64-bit Windows, run the file *curx64.exe*.

   For Unix, run the file *cur*.
3. Follow the on-screen instructions.

   The various commands manipulate a database cursor and print the query results on the screen. Enter the letter of the command that you want to perform. Some systems may require you to press Enter after the letter.

To run the dynamic cursor sample program

1. For 32-bit Windows, run the file *dcurwin.exe*.

   For 64-bit Windows, run the file *dcurx64.exe*.

   For Unix, run the file *dcur*.
2. Each sample program presents a console-type user interface and prompts you for a command. Enter the following connection string to connect to the sample database:

   ```
   DSN=SQL Anywhere 11 Demo
   ```
3. Each sample program prompts you for a table. Choose one of the tables in the sample database. For example, you can enter **Customers** or **Employees**.
4. Follow the on-screen instructions.

   The various commands manipulate a database cursor and print the query results on the screen. Enter the letter of the command you want to perform. Some systems may require you to press Enter after the letter.

Windows samples

The Windows versions of the example programs use the Windows graphical user interface. However, to keep the user interface code relatively simple, some simplifications have been made. In particular, these applications do not repaint their Windows on WM_PAINT messages except to reprint the prompt.

Static cursor sample

This example demonstrates the use of cursors. The particular cursor used here retrieves certain information from the Employees table in the sample database. The cursor is declared statically, meaning that the actual SQL statement to retrieve the information is hard coded into the source program. This is a good starting point for learning how cursors work. The Dynamic Cursor sample takes this first example and converts it to use dynamic SQL statements. See "Dynamic cursor sample" on page 516.

For information about where the source code can be found and how to build this example program, see "Sample embedded SQL programs" on page 514.

The open_cursor routine both declares a cursor for the specific SQL query and also opens the cursor.

Printing a page of information is accomplished by the print routine. It loops *pagesize* times, fetching a single row from the cursor and printing it out. Note that the fetch routine checks for warning conditions (such as `Row not found`) and prints appropriate messages when they arise. In addition, the cursor is repositioned by this program to the row before the one that appears at the top of the current page of data.

The move, top, and bottom routines use the appropriate form of the FETCH statement to position the cursor. Note that this form of the FETCH statement doesn't actually get the data—it only positions the cursor. Also, a general relative positioning routine, move, has been implemented to move in either direction depending on the sign of the parameter.

When the user quits, the cursor is closed and the database connection is also released. The cursor is closed by a ROLLBACK WORK statement, and the connection is released by a DISCONNECT.

Dynamic cursor sample

This sample demonstrates the use of cursors for a dynamic SQL SELECT statement. It is a slight modification of the static cursor example. If you have not yet looked at Static Cursor sample, it would be helpful to do so before looking at this sample. See "Static cursor sample" on page 516.

For information about where the source code can be found and how to build this sample program, see "Sample embedded SQL programs" on page 514.

The dcur program allows the user to select a table to look at with the n command. The program then presents as much information from that table as fits on the screen.

When this program is run, it prompts for a connection string of the form:

```
UID=DBA;PWD=sql;DBF=samples-dir\demo.db
```

The C program with the embedded SQL is held in the *samples-dir\SQLAnywhere\C* directory. For Windows Mobile, a dynamic cursor example is located in the *samples-dir\SQLAnywhere\CE\esql_sample* directory. The program looks much like the static cursor sample with the exception of the connect, open_cursor, and print functions.

The connect function uses the embedded SQL interface function db_string_connect to connect to the database. This function provides the extra functionality to support the connection string that is used to connect to the database.

The open_cursor routine first builds the SELECT statement

```
SELECT * FROM table-name
```

where *table-name* is a parameter passed to the routine. It then prepares a dynamic SQL statement using this string.

The embedded SQL DESCRIBE statement is used to fill in the SQLDA structure the results of the SELECT statement.

Size of the SQLDA

An initial guess is taken for the size of the SQLDA (3). If this is not big enough, the actual size of the select list returned by the database server is used to allocate a SQLDA of the correct size.

The SQLDA structure is then filled with buffers to hold strings that represent the results of the query. The fill_s_sqlda routine converts all data types in the SQLDA to DT_STRING and allocates buffers of the appropriate size.

A cursor is then declared and opened for this statement. The rest of the routines for moving and closing the cursor remain the same.

The fetch routine is slightly different: it puts the results into the SQLDA structure instead of into a list of host variables. The print routine has changed significantly to print results from the SQLDA structure up to the width of the screen. The print routine also uses the name fields of the SQLDA to print headings for each column.

Embedded SQL data types

To transfer information between a program and the database server, every piece of data must have a data type. The embedded SQL data type constants are prefixed with DT_, and can be found in the *sqldef.h* header file. You can create a host variable of any one of the supported types. You can also use these types in a SQLDA structure for passing data to and from the database.

You can define variables of these data types using the DECL_ macros listed in *sqlca.h*. For example, a variable holding a BIGINT value could be declared with DECL_BIGINT.

The following data types are supported by the embedded SQL programming interface:

- **DT_BIT** 8-bit signed integer.
- **DT_SMALLINT** 16-bit signed integer.
- **DT_UNSSMALLINT** 16-bit unsigned integer.
- **DT_TINYINT** 8-bit signed integer.
- **DT_BIGINT** 64-bit signed integer.
- **DT_UNSBIGINT** 64-bit unsigned integer.
- **DT_INT** 32-bit signed integer.
- **DT_UNSINT** 32-bit unsigned integer.
- **DT_FLOAT** 4-byte floating point number.
- **DT_DOUBLE** 8-byte floating point number.
- **DT_DECIMAL** Packed decimal number (proprietary format).

  ```
  typedef struct TYPE_DECIMAL {
    char array[1];
  } TYPE_DECIMAL;
  ```

- **DT_STRING** Null-terminated character string, in the CHAR character set. The string is blank-padded if the database is initialized with blank-padded strings.
- **DT_NSTRING** Null-terminated character string, in the NCHAR character set. The string is blank-padded if the database is initialized with blank-padded strings.
- **DT_DATE** Null-terminated character string that is a valid date.
- **DT_TIME** Null-terminated character string that is a valid time.
- **DT_TIMESTAMP** Null-terminated character string that is a valid timestamp.
- **DT_FIXCHAR** Fixed-length blank-padded character string, in the CHAR character set. The maximum length, specified in bytes, is 32767. The data is not null-terminated.
- **DT_NFIXCHAR** Fixed-length blank-padded character string, in the NCHAR character set. The maximum length, specified in bytes, is 32767. The data is not null-terminated.
- **DT_VARCHAR** Varying length character string, in the CHAR character set, with a two-byte length field. The maximum length is 32765 bytes . When sending data, you must set the length field. When fetching data, the database server sets the length field. The data is not null-terminated or blank-padded.


```
typedef struct VARCHAR {
   unsigned short int len;
   char               array[1];
} VARCHAR;
```

- **DT_NVARCHAR** Varying length character string, in the NCHAR character set, with a two-byte length field. The maximum length is 32765 bytes. When sending data, you must set the length field. When fetching data, the database server sets the length field. The data is not null-terminated or blank-padded.

  ```
  typedef struct NVARCHAR {
     unsigned short int len;
     char               array[1];
  } NVARCHAR;
  ```

- **DT_LONGVARCHAR** Long varying length character string, in the CHAR character set.

  ```
  typedef struct LONGVARCHAR {
   a_sql_uint32 array_len;  /* number of allocated bytes in array */
   a_sql_uint32 stored_len; /* number of bytes stored in array
                             * (never larger than array_len) */
   a_sql_uint32 untrunc_len;/* number of bytes in untruncated expression
                             * (may be larger than array_len) */
   char  array[1];          /* the data */
  } LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
  ```

  The LONGVARCHAR structure can be used with more than 32767 bytes of data. Large data can be fetched all at once, or in pieces using the GET DATA statement. Large data can be supplied to the server all at once, or in pieces by appending to a database variable using the SET statement. The data is not null-terminated or blank-padded.

  For more information, see "Sending and retrieving long values" on page 558.

- **DT_LONGNVARCHAR** Long varying length character string, in the NCHAR character set. The macro defines a structure, as follows:

  ```
  typedef struct LONGVARCHAR {
   a_sql_uint32 array_len;  /* number of allocated bytes in array */
   a_sql_uint32 stored_len; /* number of bytes stored in array
                             * (never larger than array_len) */
   a_sql_uint32 untrunc_len;/* number of bytes in untruncated expression
                             * (may be larger than array_len) */
   char  array[1];          /* the data */
  } LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
  ```

  The LONGNVARCHAR structure can be used with more than 32767 bytes of data. Large data can be fetched all at once, or in pieces using the GET DATA statement. Large data can be supplied to the server all at once, or in pieces by appending to a database variable using the SET statement. The data is not null-terminated or blank-padded.

  For more information, see "Sending and retrieving long values" on page 558.

- **DT_BINARY** Varying length binary data with a two-byte length field. The maximum length is 32765 bytes. When supplying information to the database server, you must set the length field. When fetching information from the database server, the server sets the length field.

  ```
  typedef struct BINARY {
    unsigned short int len;
    char array[1];
  } BINARY;
  ```

- **DT_LONGBINARY** Long binary data. The macro defines a structure, as follows:

```
typedef struct LONGVARCHAR {
 a_sql_uint32 array_len;  /* number of allocated bytes in array */
 a_sql_uint32 stored_len; /* number of bytes stored in array
                           * (never larger than array_len) */
 a_sql_uint32 untrunc_len;/* number of bytes in untruncated expression
                           * (may be larger than array_len) */
 char  array[1];          /* the data */
} LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
```

  The LONGBINARY structure may be used with more than 32767 bytes of data. Large data can be fetched all at once, or in pieces using the GET DATA statement. Large data can be supplied to the server all at once, or in pieces by appending to a database variable using the SET statement.

  For more information, see "Sending and retrieving long values" on page 558.

- **DT_TIMESTAMP_STRUCT** SQLDATETIME structure with fields for each part of a timestamp.

```
typedef struct sqldatetime {
 unsigned short year; /* for example 1999 */
 unsigned char month; /* 0-11 */
 unsigned char day_of_week; /* 0-6 0=Sunday */
 unsigned short day_of_year; /* 0-365 */
 unsigned char day; /* 1-31 */
 unsigned char hour; /* 0-23 */
 unsigned char minute; /* 0-59 */
 unsigned char second; /* 0-59 */
 unsigned long microsecond; /* 0-999999 */
} SQLDATETIME;
```

  The SQLDATETIME structure can be used to retrieve fields of DATE, TIME, and TIMESTAMP type (or anything that can be converted to one of these). Often, applications have their own formats and date manipulation code. Fetching data in this structure makes it easier for a programmer to manipulate this data. Note that DATE, TIME, and TIMESTAMP fields can also be fetched and updated with any character type.

  If you use a SQLDATETIME structure to enter a date, time, or timestamp into the database, the day_of_year and day_of_week members are ignored.

  See:

  - "date_format option [compatibility]" [*SQL Anywhere Server - Database Administration*]
  - "date_order option [compatibility]" [*SQL Anywhere Server - Database Administration*]
  - "time_format option [compatibility]" [*SQL Anywhere Server - Database Administration*]
  - "timestamp_format option [compatibility]" [*SQL Anywhere Server - Database Administration*]

- **DT_VARIABLE** Null-terminated character string. The character string must be the name of a SQL variable whose value is used by the database server. This data type is used only for supplying data to the database server. It cannot be used when fetching data from the database server.

The structures are defined in the *sqlca.h* file. VARCHAR, NVARCHAR, BINARY, DECIMAL, and the LONG data types contain a one-character array and are thus not useful for declaring host variables but they are useful for allocating variables dynamically or typecasting other variables.

DATE and TIME database types

There are no corresponding embedded SQL interface data types for the various DATE and TIME database types. These database types are all fetched and updated using either the SQLDATETIME structure or character strings.

For more information, see "GET DATA statement [ESQL]" [*SQL Anywhere Server - SQL Reference*] and "SET statement" [*SQL Anywhere Server - SQL Reference*].

Using host variables

Host variables are C variables that are identified to the SQL preprocessor. Host variables can be used to send values to the database server or receive values from the database server.

Host variables are quite easy to use, but they have some restrictions. Dynamic SQL is a more general way of passing information to and from the database server using a structure known as the SQL Descriptor Area (SQLDA). The SQL preprocessor automatically generates a SQLDA for each statement in which host variables are used.

You cannot use host variables in batches.

For information about dynamic SQL, see "Static and dynamic SQL" on page 537.

Declaring host variables

Host variables are defined by putting them into a **declaration section**. According to the ANSI embedded SQL standard, host variables are defined by surrounding the normal C variable declarations with the following:

```
EXEC SQL BEGIN DECLARE SECTION;
/* C variable declarations */
EXEC SQL END DECLARE SECTION;
```

These host variables can then be used in place of value constants in any SQL statement. When the database server executes the statement, the value of the host variable is used. Note that host variables cannot be used in place of table or column names: dynamic SQL is required for this. The variable name is prefixed with a colon (:) in a SQL statement to distinguish it from other identifiers allowed in the statement.

The SQL preprocessor does not scan C language code except inside a DECLARE SECTION. Thus, TYPEDEF types and structures are not allowed. Initializers on the variables are allowed inside a DECLARE SECTION.

Example

The following sample code illustrates the use of host variables on an INSERT statement. The variables are filled in by the program and then inserted into the database:

```
EXEC SQL BEGIN DECLARE SECTION;
long employee_number;
char employee_name[50];
char employee_initials[8];
char employee_phone[15];
EXEC SQL END DECLARE SECTION;
/* program fills in variables with appropriate values
*/
EXEC SQL INSERT INTO Employees
VALUES (:employee_number, :employee_name,
:employee_initials, :employee_phone );
```

For a more extensive example, see "Static cursor sample" on page 516.

C host variable types

Only a limited number of C data types are supported as host variables. Also, certain host variable types do not have a corresponding C type.

Macros defined in the *sqlca.h* header file can be used to declare host variables of the following types: NCHAR, VARCHAR, NVARCHAR, LONGVARCHAR, LONGNVARCHAR, BINARY, LONGBINARY, DECIMAL, FIXCHAR, NFIXCHAR, DATETIME (SQLDATETIME), BIT, BIGINT, or UNSIGNED BIGINT. They are used as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_NCHAR                 v_nchar[10];
DECL_VARCHAR( 10 )         v_varchar;
DECL_NVARCHAR( 10 )        v_nvarchar;
DECL_LONGVARCHAR( 32768 )  v_longvarchar;
DECL_LONGNVARCHAR( 32768 ) v_longnvarchar;
DECL_BINARY( 4000 )        v_binary;
DECL_LONGBINARY( 128000 )  v_longbinary;
DECL_DECIMAL( 30, 6 )      v_decimal;
DECL_FIXCHAR( 10 )         v_fixchar;
DECL_NFIXCHAR( 10 )        v_nfixchar;
DECL_DATETIME              v_datetime;
DECL_BIT                   v_bit;
DECL_BIGINT                v_bigint;
DECL_UNSIGNED_BIGINT       v_ubigint;
EXEC SQL END DECLARE SECTION;
```

The preprocessor recognizes these macros within a declaration section and treats the variable as the appropriate type. It is recommended that the DECIMAL (DT_DECIMAL, DECL_DECIMAL) type not be used since the format of decimal numbers is proprietary.

The following table lists the C variable types that are allowed for host variables and their corresponding embedded SQL interface data types.

C data type	Embedded SQL interface type	Description
`short si;` `short int si;`	DT_SMALLINT	16-bit signed integer.
`unsigned short int usi;`	DT_UNSSMALLINT	16-bit unsigned integer.
`long l;` `long int l;`	DT_INT	32-bit signed integer.
`unsigned long int ul;`	DT_UNSINT	32-bit unsigned integer.
`DECL_BIGINT ll;`	DT_BIGINT	64-bit signed integer.
`DECL_UNSIGNED_BIGINT ull;`	DT_UNSBIGINT	64-bit unsigned integer.
`float f;`	DT_FLOAT	4-byte floating point.
`double d;`	DT_DOUBLE	8-byte floating point.

C data type	Embedded SQL interface type	Description
`char a[n]; /*n>=1*/`	DT_STRING	Null-terminated string, in CHAR character set. The string is blank-padded if the database is initialized with blank-padded strings. This variable holds n-1 bytes plus the null terminator.
`char *a;`	DT_STRING	Null-terminated string, in CHAR character set. This variable points to an area that can hold up to 32766 bytes plus the null terminator.
`DECL_NCHAR a[n]; /*n>=1*/`	DT_NSTRING	Null-terminated string, in NCHAR character set. The string is blank-padded if the database is initialized with blank-padded strings. This variable holds n-1 bytes plus the null terminator.
`DECL_NCHAR *a;`	DT_NSTRING	Null-terminated string, in NCHAR character set. This variable points to an area that can hold up to 32766 bytes plus the null terminator.
`DECL_VARCHAR(n) a;`	DT_VARCHAR	Varying length character string, in CHAR character set, with 2-byte length field. Not null-terminated or blank-padded. The maximum value for n is 32765 (bytes).
`DECL_NVARCHAR(n) a;`	DT_NVARCHAR	Varying length character string, in NCHAR character set, with 2-byte length field. Not null-terminated or blank-padded. The maximum value for n is 32765 (bytes).

C data type	Embedded SQL interface type	Description
`DECL_LONGVARCHAR(n) a;`	DT_LONGVARCHAR	Varying length long character string, in CHAR character set, with three 4-byte length fields. Not null-terminated or blank-padded.
`DECL_LONGNVARCHAR(n) a;`	DT_LONGNVARCHAR	Varying length long character string, in NCHAR character set, with three 4-byte length fields. Not null-terminated or blank-padded.
`DECL_BINARY(n) a;`	DT_BINARY	Varying length binary data with 2-byte length field. The maximum value for n is 32765 (bytes).
`DECL_LONGBINARY(n) a;`	DT_LONGBINARY	Varying length long binary data with three 4-byte length fields.
`char a; /*n=1*/` `DECL_FIXCHAR(n) a;`	DT_FIXCHAR	Fixed length character string, in CHAR character set. Blank-padded but not null-terminated. The maximum value for n is 32767 (bytes).
`DECL_NCHAR a; /*n=1*/` `DECL_NFIXCHAR(n) a;`	DT_NFIXCHAR	Fixed length character string, in NCHAR character set. Blank-padded but not null-terminated. The maximum value for n is 32767 (bytes).
`DECL_DATETIME a;`	DT_TIME-STAMP_STRUCT	SQLDATETIME structure

Character sets

For DT_FIXCHAR, DT_STRING, DT_VARCHAR, and DT_LONGVARCHAR, character data is in the application's CHAR character set, which is usually the character set of the application's locale. An application can change the CHAR character set either by using the CHARSET connection parameter, or by calling the db_change_char_charset function.

For DT_NFIXCHAR, DT_NSTRING, DT_NVARCHAR, and DT_LONGNVARCHAR, data is in the application's NCHAR character set. By default, the application's NCHAR character set is the same as the

CHAR character set. An application can change the NCHAR character set by calling the db_change_nchar_charset function.

For more information about locales and character sets, see "Understanding locales" [*SQL Anywhere Server - Database Administration*].

For more information about changing the CHAR character set, see "CharSet connection parameter [CS]" [*SQL Anywhere Server - Database Administration*] or "db_change_char_charset function" on page 574.

For more information about changing the NCHAR character set, see "db_change_nchar_charset function" on page 574.

Data lengths

Regardless of the CHAR and NCHAR character sets in use, all data lengths are specified in bytes.

If character set conversion occurs between the server and the application, it is the application's responsibility to ensure that buffers are sufficiently large to handle the converted data, and to issue additional GET DATA statements if data is truncated.

Pointers to char

The database interface considers a host variable declared as a **pointer to char** (*char * a*) to be 32767 bytes long. Any host variable of type pointer to char used to retrieve information from the database must point to a buffer large enough to hold any value that could possibly come back from the database.

This is potentially quite dangerous because someone could change the definition of the column in the database to be larger than it was when the program was written. This could cause random memory corruption problems. It is better to use a declared array, even as a parameter to a function, where it is passed as a pointer to char. This technique allows the embedded SQL statements to know the size of the array.

Scope of host variables

A standard host-variable declaration section can appear anywhere that C variables can normally be declared. This includes the parameter declaration section of a C function. The C variables have their normal scope (available within the block in which they are defined). However, since the SQL preprocessor does not scan C code, it does not respect C blocks.

As far as the SQL preprocessor is concerned, host variables are global to the source file; two host variables cannot have the same name.

Host variable usage

Host variables can be used in the following circumstances:

- SELECT, INSERT, UPDATE, and DELETE statements in any place where a number or string constant is allowed.
- The INTO clause of SELECT and FETCH statements.

- Host variables can also be used in place of a statement name, a cursor name, or an option name in statements specific to embedded SQL.
- For CONNECT, DISCONNECT, and SET CONNECT statements, a host variable can be used in place of a server name, database name, connection name, user ID, password, or connection string.
- For SET OPTION and GET OPTION, a host variable can be used in place of a user ID, option name, or option value.
- Host variables cannot be used in place of a table name or a column name in any statement.

SQLCODE and SQLSTATE host variables

The ISO/ANSI standard allows an embedded SQL source file to declare the following special host variables within a declaration section:

```
long SQLCODE;
char SQLSTATE[6];
```

If used, these variables are set after any embedded SQL statement that makes a database request (EXEC SQL statements other than DECLARE SECTION, INCLUDE, WHENEVER SQLCODE, and so on).

The SQLCODE and SQLSTATE host variables must be visible in the scope of every embedded SQL statement that generates database requests.

For more information, see the description of the sqlpp -k option in "SQL preprocessor" on page 566.

The following is valid embedded SQL:

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
long SQLCODE;
EXEC SQL END DECLARE SECTION;
sub1() {
 EXEC SQL BEGIN DECLARE SECTION;
 char SQLSTATE[6];
 EXEC SQL END DECLARE SECTION;
 exec SQL CREATE TABLE ...
}
```

The following is not valid embedded SQL:

```
EXEC SQL INCLUDE SQLCA;
sub1() {
 EXEC SQL BEGIN DECLARE SECTION;
 char SQLSTATE[6];
 EXEC SQL END DECLARE SECTION;
 exec SQL CREATE TABLE...
}
sub2() {
 exec SQL DROP TABLE...
 // No SQLSTATE in scope of this statement
}
```

Indicator variables

Indicator variables are C variables that hold supplementary information when you are fetching or putting data. There are several distinct uses for indicator variables:

- **NULL values** To enable applications to handle NULL values.
- **String truncation** To enable applications to handle cases when fetched values must be truncated to fit into host variables.
- **Conversion errors** To hold error information.

An indicator variable is a host variable of type short int that is placed immediately following a regular host variable in a SQL statement. For example, in the following INSERT statement, :ind_phone is an indicator variable:

```
EXEC SQL INSERT INTO Employees
 VALUES (:employee_number, :employee_name,
 :employee_initials, :employee_phone:ind_phone );
```

On a fetch or execute where no rows are received from the database server (such as when an error or end of result set occurs), then indicator values are unchanged.

Using indicator variables to handle NULL

Do not confuse the SQL concept of NULL with the C-language constant of the same name. In the SQL language, NULL represents either an unknown attribute or inapplicable information. The C-language constant represents a pointer value that does not point to a memory location.

When NULL is used in the SQL Anywhere documentation, it refers to the SQL database meaning given above. The C language constant is referred to as the null pointer (lowercase).

NULL is not the same as any value of the column's defined type. Thus, to pass NULL values to the database or receive NULL results back, something extra is required beyond regular host variables. **Indicator variables** are used for this purpose.

Using indicator variables when inserting NULL

An INSERT statement could include an indicator variable as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
short int employee_number;
char employee_name[50];
char employee_initials[6];
char employee_phone[15];
short int ind_phone;
EXEC SQL END DECLARE SECTION;
/*
This program fills in the employee number,
name, initials, and phone number.
*/
if( /* Phone number is unknown */ ) {
 ind_phone = -1;
} else {
 ind_phone = 0;
}
EXEC SQL INSERT INTO Employees
 VALUES (:employee_number, :employee_name,
 :employee_initials, :employee_phone:ind_phone );
```


If the indicator variable has a value of -1, a NULL is written. If it has a value of 0, the actual value of employee_phone is written.

Using indicator variables when fetching NULL

Indicator variables are also used when receiving data from the database. They are used to indicate that a NULL value was fetched (indicator is negative). If a NULL value is fetched from the database and an indicator variable is not supplied, an error is generated (SQLE_NO_INDICATOR).

Using indicator variables for truncated values

Indicator variables indicate whether any fetched values were truncated to fit into a host variable. This enables applications to handle truncation appropriately.

If a value is truncated on fetching, the indicator variable is set to a positive value, containing the actual length of the database value before truncation. If the length of the value is greater than 32767 bytes, then the indicator variable contains 32767.

Using indicator values for conversion errors

By default, the conversion_error database option is set to On, and any data type conversion failure leads to an error, with no row returned.

You can use indicator variables to tell which column produced a data type conversion failure. If you set the database option conversion_error to Off, any data type conversion failure gives a CANNOT_CONVERT warning, rather than an error. If the column that suffered the conversion error has an indicator variable, that variable is set to a value of -2.

If you set the conversion_error option to Off when inserting data into the database, a value of NULL is inserted when a conversion failure occurs.

Summary of indicator variable values

The following table provides a summary of indicator variable usage.

Indicator value	Supplying value to database	Receiving value from database
> 0	Host variable value	Retrieved value was truncated—actual length in indicator variable
0	Host variable value	Fetch successful, or conversion_error set to On
-1	NULL value	NULL result

Indicator value	Supplying value to database	Receiving value from database
-2	NULL value	Conversion error (when conversion_error is set to Off only). SQLCODE indicates a CANNOT_CONVERT warning
< -2	NULL value	NULL result

For more information about retrieving long values, see "GET DATA statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

The SQL Communication Area (SQLCA)

The **SQL Communication Area** (**SQLCA**) is an area of memory that is used on every database request for communicating statistics and errors from the application to the database server and back to the application. The SQLCA is used as a handle for the application-to-database communication link. It is passed in to all database library functions that need to communicate with the database server. It is implicitly passed on all embedded SQL statements.

A global SQLCA variable is defined in the interface library. The preprocessor generates an external reference for the global SQLCA variable and an external reference for a pointer to it. The external reference is named sqlca and is of type SQLCA. The pointer is named sqlcaptr. The actual global variable is declared in the imports library.

The SQLCA is defined by the *sqlca.h* header file, included in the *SDK\Include* subdirectory of your installation directory.

SQLCA provides error codes

You reference the SQLCA to test for a particular error code. The sqlcode and sqlstate fields contain error codes when a database request has an error. Some C macros are defined for referencing the sqlcode field, the sqlstate field, and some other fields.

SQLCA fields

The fields in the SQLCA have the following meanings:

- **sqlcaid** An 8-byte character field that contains the string SQLCA as an identification of the SQLCA structure. This field helps in debugging when you are looking at memory contents.
- **sqlcabc** A long integer that contains the length of the SQLCA structure (136 bytes).
- **sqlcode** A long integer that specifies the error code when the database detects an error on a request. Definitions for the error codes can be found in the header file *sqlerr.h*. The error code is 0 (zero) for a successful operation, positive for a warning, and negative for an error.

  For a full listing of error codes, see Error Messages [*Error Messages*].
- **sqlerrml** The length of the information in the sqlerrmc field.
- **sqlerrmc** Zero or more character strings to be inserted into an error message. Some error messages contain one or more placeholder strings (*%1*, *%2*, ...) that are replaced with the strings in this field.

  For example, if a `Table Not Found` error is generated, sqlerrmc contains the table name, which is inserted into the error message at the appropriate place.

  For a full listing of error messages, see Error Messages [*Error Messages*].
- **sqlerrp** Reserved.
- **sqlerrd** A utility array of long integers.
- **sqlwarn** Reserved.

- **sqlstate** The SQLSTATE status value. The ANSI SQL standard defines this type of return value from a SQL statement in addition to the SQLCODE value. The SQLSTATE value is always a five-character null-terminated string, divided into a two-character class (the first two characters) and a three-character subclass. Each character can be a digit from 0 through 9 or an uppercase alphabetic character A through Z.

  Any class or subclass that begins with 0 through 4 or A through H is defined by the SQL standard; other classes and subclasses are implementation defined. The SQLSTATE value '00000' means that there has been no error or warning.

  For more SQLSTATE values, see Error Messages [*Error Messages*].

sqlerror array

The sqlerror field array has the following elements.

- **sqlerrd[1] (SQLIOCOUNT)** The actual number of input/output operations that were required to complete a statement.

  The database server does not set this number to zero for each statement. Your program can set this variable to zero before executing a sequence of statements. After the last statement, this number is the total number of input/output operations for the entire statement sequence.

- **sqlerrd[2] (SQLCOUNT)** The value of this field depends on which statement is being executed.
  - **INSERT, UPDATE, PUT, and DELETE statements** The number of rows that were affected by the statement.
  - **OPEN statement** On a cursor OPEN, this field is filled in with either the actual number of rows in the cursor (a value greater than *or equal to* 0) or an estimate thereof (a negative number whose absolute value is the estimate). It is the actual number of rows if the database server can compute it without counting the rows. The database can also be configured to always return the actual number of rows using the row_counts option.
  - **FETCH cursor statement** The SQLCOUNT field is filled if a SQLE_NOTFOUND warning is returned. It contains the number of rows by which a FETCH RELATIVE or FETCH ABSOLUTE statement goes outside the range of possible cursor positions (a cursor can be on a row, before the first row, or after the last row). In the case of a wide fetch, SQLCOUNT is the number of rows actually fetched, and is less than or equal to the number of rows requested. During a wide fetch, SQLE_NOTFOUND is only set if no rows are returned.

    For more information about wide fetches, see "Fetching more than one row at a time" on page 553.

    The value is 0 if the row was not found, but the position is valid, for example, executing FETCH RELATIVE 1 when positioned on the last row of a cursor. The value is positive if the attempted fetch was beyond the end of the cursor, and negative if the attempted fetch was before the beginning of the cursor.
  - **GET DATA statement** The SQLCOUNT field holds the actual length of the value.
  - **DESCRIBE statement** In the WITH VARIABLE RESULT clause used to describe procedures that may have more than one result set, SQLCOUNT is set to one of the following values:

- **0** The result set may change: the procedure call should be described again following each OPEN statement.
- **1** The result set is fixed. No re-describing is required.

In the case of a syntax error, SQLE_SYNTAX_ERROR, this field contains the approximate character position within the statement where the error was detected.

- **sqlerrd[3] (SQLIOESTIMATE)** The estimated number of input/output operations that are required to complete the statement. This field is given a value on an OPEN or EXPLAIN statement.

SQLCA management for multi-threaded or reentrant code

You can use embedded SQL statements in multi-threaded or reentrant code. However, if you use a single connection, you are restricted to one active request per connection. In a multi-threaded application, you should not use the same connection to the database on each thread unless you use a semaphore to control access.

There are no restrictions on using separate connections on each thread that wants to use the database. The SQLCA is used by the runtime library to distinguish between the different thread contexts. Thus, each thread wanting to use the database concurrently must have its own SQLCA.

Any given database connection is accessible only from one SQLCA, with the exception of the cancel instruction, which must be issued from a separate thread.

For information about canceling requests, see "Implementing request management" on page 565.

The following is an example of multi-threaded embedded SQL reentrant code.

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <ctype.h>
#include <stdlib.h>
#include <process.h>
#include <windows.h>
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;

#define TRUE 1
#define FALSE 0

// multithreading support

typedef struct a_thread_data {
    SQLCA sqlca;
    int num_iters;
    int thread;
    int done;
} a_thread_data;

// each thread's ESQL test

EXEC SQL SET SQLCA "&thread_data->sqlca";

static void PrintSQLError( a_thread_data * thread_data )
/*****************************************************/
```

```
{
  char                buffer[200];

  printf( "%d: SQL error %d -- %s ... aborting\n",
      thread_data->thread,
      SQLCODE,
      sqlerror_message( &thread_data->sqlca,
                buffer, sizeof( buffer ) ) );
    exit( 1 );
}

EXEC SQL WHENEVER SQLERROR { PrintSQLError( thread_data ); };

static void do_one_iter( void * data )
{
  a_thread_data *  thread_data = (a_thread_data *)data;
  int    i;
  EXEC SQL BEGIN DECLARE SECTION;
  char    user[ 20 ];
  EXEC SQL END DECLARE SECTION;

  if( db_init( &thread_data->sqlca ) != 0 ) {
    for( i = 0; i < thread_data->num_iters; i++ ) {
     EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";
     EXEC SQL SELECT USER INTO :user;
     EXEC SQL DISCONNECT;
    }
    printf( "Thread %d did %d iters successfully\n",
        thread_data->thread, thread_data->num_iters );
    db_fini( &thread_data->sqlca );
  }
  thread_data->done = TRUE;
}

int main()
{
  int num_threads = 4;
  int thread;
  int num_iters = 300;
  int num_done = 0;
  a_thread_data *thread_data;
  thread_data = (a_thread_data *)malloc( sizeof( a_thread_data ) *
num_threads );
  for( thread = 0; thread < num_threads; thread++ ) {
    thread_data[ thread ].num_iters = num_iters;
    thread_data[ thread ].thread = thread;
    thread_data[ thread ].done = FALSE;
    if( _beginthread( do_one_iter,
      8096,
      (void *)&thread_data[thread] ) <= 0 ) {
      printf( "FAILED creating thread.\n" );
      return( 1 );
    }
  }
    while( num_done != num_threads ) {
    Sleep( 1000 );
    num_done = 0;
    for( thread = 0; thread < num_threads; thread++ ) {
      if( thread_data[ thread ].done == TRUE ) {
        num_done++;
      }
    }
  }
```



```
    return( 0 );
}
```

Using multiple SQLCAs

To manage multiple SQLCAs in your application

1. You must not use the option on the SQL preprocessor that generates non-reentrant code (-r-). The reentrant code is a little larger and a little slower because statically initialized global variables cannot be used. However, these effects are minimal.
2. Each SQLCA used in your program must be initialized with a call to db_init and cleaned up at the end with a call to db_fini.
3. The embedded SQL statement SET SQLCA is used to tell the SQL preprocessor to use a different SQLCA for database requests. Usually, a statement such as: `EXEC SQL SET SQLCA 'task_data->sqlca';` is used at the top of your program or in a header file to set the SQLCA reference to point at task specific data. This statement does not generate any code and thus has no performance impact. It changes the state within the preprocessor so that any reference to the SQLCA uses the given string.

For information about creating SQLCAs, see "SET SQLCA statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

When to use multiple SQLCAs

You can use the multiple SQLCA support in any of the supported embedded SQL environments, but it is only required in reentrant code.

The following list details the environments where multiple SQLCAs must be used:

- **Multi-threaded applications** Each thread must have its own SQLCA. This can also happen when you have a DLL that uses embedded SQL and is called by more than one thread in your application.
- **Dynamic link libraries and shared libraries** A DLL has only one data segment. While the database server is processing a request from one application, it may yield to another application that makes a request to the database server. If your DLL uses the global SQLCA, both applications are using it at the same time. Each Windows application must have its own SQLCA.
- **A DLL with one data segment** A DLL can be created with only one data segment or one data segment for each application. If your DLL has only one data segment, you cannot use the global SQLCA for the same reason that a DLL cannot use the global SQLCA. Each application must have its own SQLCA.

Connection management with multiple SQLCAs

You do not need to use multiple SQLCAs to connect to more than one database or have more than one connection to a single database.

Each SQLCA can have one unnamed connection. Each SQLCA has an active or current connection See "SET CONNECTION statement [Interactive SQL] [ESQL]" [*SQL Anywhere Server - SQL Reference*].

All operations on a given database connection must use the same SQLCA that was used when the connection was established.

Record locking
Operations on different connections are subject to the normal record locking mechanisms and may cause each other to block and possibly to deadlock. For information about locking, see "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*].

Static and dynamic SQL

There are two ways to embed SQL statements into a C program:

- Static statements
- Dynamic statements

Until now, static SQL has been discussed. This section compares static and dynamic SQL.

Static SQL statements

All standard SQL data manipulation and data definition statements can be embedded in a C program by prefixing them with EXEC SQL and suffixing the statement with a semicolon (;). These statements are referred to as **static** statements.

Static statements can contain references to host variables. All examples to this point have used static embedded SQL statements. See "Using host variables" on page 522.

Host variables can only be used in place of string or numeric constants. They cannot be used to substitute column names or table names; dynamic statements are required to perform those operations.

Dynamic SQL statements

In the C language, strings are stored in arrays of characters. Dynamic statements are constructed in C language strings. These statements can then be executed using the PREPARE and EXECUTE statements. These SQL statements cannot reference host variables in the same manner as static statements since the C language variables are not accessible by name when the C program is executing.

To pass information between the statements and the C language variables, a data structure called the **SQL Descriptor Area** (**SQLDA**) is used. This structure is set up for you by the SQL preprocessor if you specify a list of host variables on the EXECUTE statement in the USING clause. These variables correspond by position to place holders in the appropriate positions of the prepared statement.

For information about the SQLDA, see "The SQL descriptor area (SQLDA)" on page 541.

A **place holder** is put in the statement to indicate where host variables are to be accessed. A place holder is either a question mark (?) or a host variable reference as in static statements (a host variable name preceded by a colon). In the latter case, the host variable name used in the actual text of the statement serves only as a place holder indicating a reference to the SQL descriptor area.

A host variable used to pass information to the database is called a **bind variable**.

Example

For example:

```
EXEC SQL BEGIN DECLARE SECTION;
char comm[200];
char Street[30];
```

```
char City[20];
short int cityind;
long empnum;
EXEC SQL END DECLARE SECTION;

...
sprintf( comm,
    "UPDATE %s SET Street = :?, City = :?"
    "WHERE employee_number = :?",
    tablename );
EXEC SQL PREPARE S1 FROM :comm;
EXEC SQL EXECUTE S1 USING :Street, :City:cityind, :empnum;
```

This method requires you to know how many host variables there are in the statement. Usually, this is not the case. So, you can set up your own SQLDA structure and specify this SQLDA in the USING clause on the EXECUTE statement.

The DESCRIBE BIND VARIABLES statement returns the host variable names of the bind variables that are found in a prepared statement. This makes it easier for a C program to manage the host variables. The general method is as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
char comm[200];
EXEC SQL END DECLARE SECTION;
...
sprintf( comm, "update %s set Street = :Street,
    City = :City"
   " where EmployeeNumber = :empnum",
   tablename );
EXEC SQL PREPARE S1 FROM :comm;
/* Assume that there are no more than 10 host variables.
 * See next example if you cannot put a limit on it. */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE BIND VARIABLES FOR S1 INTO sqlda;
/* sqlda->sqld will tell you how many
   host variables there were. */
/* Fill in SQLDA_VARIABLE fields with
   values based on name fields in sqlda. */
...
EXEC SQL EXECUTE S1 USING DESCRIPTOR sqlda;
free_sqlda( sqlda );
```

SQLDA contents

The SQLDA consists of an array of variable descriptors. Each descriptor describes the attributes of the corresponding C program variable or the location that the database stores data into or retrieves data from:

- data type
- length if *type* is a string type
- memory address
- indicator variable

For a complete description of the SQLDA structure, see "The SQL descriptor area (SQLDA)" on page 541.

Indicator variables and NULL

The indicator variable is used to pass a NULL value to the database or retrieve a NULL value from the database. The database server also uses the indicator variable to indicate truncation conditions encountered

during a database operation. The indicator variable is set to a positive value when not enough space was provided to receive a database value.

For more information, see "Indicator variables" on page 527.

Dynamic SELECT statement

A SELECT statement that returns only a single row can be prepared dynamically, followed by an EXECUTE with an INTO clause to retrieve the one-row result. SELECT statements that return multiple rows, however, are managed using dynamic cursors.

With dynamic cursors, results are put into a host variable list or a SQLDA that is specified on the FETCH statement (FETCH INTO and FETCH USING DESCRIPTOR). Since the number of select list items is usually unknown to the C programmer, the SQLDA route is the most common. The DESCRIBE SELECT LIST statement sets up a SQLDA with the types of the select list items. Space is then allocated for the values using the fill_sqlda or fill_s_sqlda functions, and the information is retrieved by the FETCH USING DESCRIPTOR statement.

The typical scenario is as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
char comm[200];
EXEC SQL END DECLARE SECTION;
int actual_size;
SQLDA * sqlda;
...
sprintf( comm, "select * from %s", table_name );
EXEC SQL PREPARE S1 FROM :comm;
/* Initial guess of 10 columns in result.
   If it is wrong, it is corrected right
   after the first DESCRIBE by reallocating
   sqlda and doing DESCRIBE again. */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE SELECT LIST FOR S1
  INTO sqlda;
if( sqlda->sqld > sqlda->sqln )
{
  actual_size = sqlda->sqld;
  free_sqlda( sqlda );
  sqlda = alloc_sqlda( actual_size );
  EXEC SQL DESCRIBE SELECT LIST FOR S1
    INTO sqlda;
}
fill_sqlda( sqlda );
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
EXEC SQL WHENEVER NOTFOUND {break};
for( ;; )
{
  EXEC SQL FETCH C1 USING DESCRIPTOR sqlda;
  /* do something with data */
}
EXEC SQL CLOSE C1;
EXEC SQL DROP STATEMENT S1;
```

Drop statements after use
To avoid consuming unnecessary resources, ensure that statements are dropped after use.

For a complete example using cursors for a dynamic select statement, see "Dynamic cursor sample" on page 516.

For more information about the functions mentioned above, see "Library function reference" on page 569.

The SQL descriptor area (SQLDA)

The SQLDA (SQL Descriptor Area) is an interface structure that is used for dynamic SQL statements. The structure passes information regarding host variables and SELECT statement results to and from the database. The SQLDA is defined in the header file *sqlda.h*.

There are functions in the database interface library or DLL that you can use to manage SQLDAs. For descriptions, see "Library function reference" on page 569.

When host variables are used with static SQL statements, the preprocessor constructs a SQLDA for those host variables. It is this SQLDA that is actually passed to and from the database server.

The SQLDA header file

The contents of *sqlda.h* are as follows:

```
#ifndef _SQLDA_H_INCLUDED
#define _SQLDA_H_INCLUDED
#define II_SQLDA

#include "sqlca.h"

#if defined( _SQL_PACK_STRUCTURES )
#include "pshpk1.h"
#endif

#define SQL_MAX_NAME_LEN  30

#define _sqldafar
typedef short int   a_sql_type;
struct sqlname
{
  short int length; /* length of char data */
  char  data[ SQL_MAX_NAME_LEN ]; /* data */
};
struct sqlvar
{   /* array of variable descriptors */
  short int  sqltype; /* type of host variable */
  short int  sqllen;  /* length of host variable */
  void      *sqldata; /* address of variable */
  short int *sqlind;  /* indicator variable pointer */
  struct sqlname  sqlname;
};
struct sqlda
{
  unsigned char sqldaid[8]; /* eye catcher "SQLDA" */
  a_sql_int32   sqldabc; /* length of sqlda structure */
  short int     sqln;    /* descriptor size in number of entries */
  short int     sqld;    /* number of variables found by DESCRIBE */
  struct sqlvar sqlvar[1]; /* array of variable descriptors */
};

typedef struct sqlda    SQLDA;
typedef struct sqlvar   SQLVAR, SQLDA_VARIABLE;
typedef struct sqlname  SQLNAME, SQLDA_NAME;
#ifndef SQLDASIZE
#define SQLDASIZE(n)    ( sizeof( struct sqlda ) +  \
```

```
                            (n-1) * sizeof( struct sqlvar) )
#endif
#if defined( _SQL_PACK_STRUCTURES )
#include "poppk.h"
#endif
#endif
```

SQLDA fields

The SQLDA fields have the following meanings:

Field	Description
sqldaid	An 8-byte character field that contains the string **SQLDA** as an identification of the SQLDA structure. This field helps in debugging when you are looking at memory contents.
sqldabc	A long integer containing the length of the SQLDA structure.
sqln	The number of variable descriptors allocated in the sqlvar array.
sqld	The number of variable descriptors that are valid (contain information describing a host variable). This field is set by the DESCRIBE statement and sometimes by the programmer when supplying data to the database server.
sqlvar	An array of descriptors of type struct sqlvar, each describing a host variable.

SQLDA host variable descriptions

Each sqlvar structure in the SQLDA describes a host variable. The fields of the sqlvar structure have the following meanings:

- **sqltype** The type of the variable that is described by this descriptor. See "Embedded SQL data types" on page 518.

  The low order bit indicates whether NULL values are allowed. Valid types and constant definitions can be found in the *sqldef.h* header file.

  This field is filled by the DESCRIBE statement. You can set this field to any type when supplying data to the database server or retrieving data from the database server. Any necessary type conversion is done automatically.

- **sqllen** The length of the variable. What the length actually means depends on the type information and how the SQLDA is being used.

  For LONG VARCHAR, LONG NVARCHAR, and LONG BINARY data types, the array_len field of the DT_LONGVARCHAR, DT_LONGNVARCHAR, or DT_LONGBINARY data type structure is used instead of the sqllen field.

For more information about the length field, see "SQLDA sqllen field values" on page 544.

- **sqldata** A pointer to the memory occupied by this variable. This memory must correspond to the sqltype and sqllen fields.

  For storage formats, see "Embedded SQL data types" on page 518.

  For UPDATE and INSERT statements, this variable is not involved in the operation if the sqldata pointer is a null pointer. For a FETCH, no data is returned if the sqldata pointer is a null pointer. In other words, the column returned by the sqldata pointer is an **unbound column**.

  If the DESCRIBE statement uses LONG NAMES, this field holds the long name of the result set column. If, in addition, the DESCRIBE statement is a DESCRIBE USER TYPES statement, then this field holds the long name of the user-defined data type, instead of the column. If the type is a base type, the field is empty.

- **sqlind** A pointer to the indicator value. An indicator value is a short int. A negative indicator value indicates a NULL value. A positive indicator value indicates that this variable has been truncated by a FETCH statement, and the indicator value contains the length of the data before truncation. A value of -2 indicates a conversion error if the conversion_error database option is set to Off. See "conversion_error option [compatibility]" [*SQL Anywhere Server - Database Administration*].

  For more information, see "Indicator variables" on page 527.

  If the sqlind pointer is the null pointer, no indicator variable pertains to this host variable.

  The sqlind field is also used by the DESCRIBE statement to indicate parameter types. If the type is a user-defined data type, this field is set to DT_HAS_USERTYPE_INFO. In this case, you may want to perform a DESCRIBE USER TYPES to obtain information on the user-defined data types.

- **sqlname** A VARCHAR-like structure, as follows:

  ```
  struct sqlname {
     short int  length;
     char  data[ SQL_MAX_NAME_LEN ];
  };
  ```

  It is filled by a DESCRIBE statement and is not otherwise used. This field has a different meaning for the two formats of the DESCRIBE statement:

  - **SELECT LIST** The name data buffer is filled with the column heading of the corresponding item in the select list.
  - **BIND VARIABLES** The name data buffer is filled with the name of the host variable that was used as a bind variable, or "?" if an unnamed parameter marker is used.

  On a DESCRIBE SELECT LIST statement, any indicator variables present are filled with a flag indicating whether the select list item is updatable or not. More information about this flag can be found in the *sqldef.h* header file.

  If the DESCRIBE statement is a DESCRIBE USER TYPES statement, then this field holds the long name of the user-defined data type instead of the column. If the type is a base type, the field is empty.

SQLDA sqllen field values

The sqllen field length of the sqlvar structure in a SQLDA is used in the following kinds of interactions with the database server:

- **describing values** The DESCRIBE statement gets information about the host variables required to store data retrieved from the database, or host variables required to pass data to the database. See "Describing values" on page 544.
- **retrieving values** Retrieving values from the database. See "Retrieving values" on page 547.
- **sending values** Sending information to the database. See "Sending values" on page 546.

These interactions are described in this section.

The following tables detail each of these interactions. These tables list the interface constant types (the **DT_** types) found in the *sqldef.h* header file. These constants would be placed in the SQLDA sqltype field.

For information about sqltype field values, see "Embedded SQL data types" on page 518.

In static SQL, a SQLDA is still used, but it is generated and completely filled in by the SQL preprocessor. In this static case, the tables give the correspondence between the static C language host variable types and the interface constants.

Describing values

The following table indicates the values of the sqllen and sqltype structure members returned by the DESCRIBE statement for the various database types (both SELECT LIST and BIND VARIABLE DESCRIBE statements). In the case of a user-defined database data type, the base type is described.

Your program can use the types and lengths returned from a DESCRIBE, or you may use another type. The database server performs type conversions between any two types. The memory pointed to by the sqldata field must correspond to the sqltype and sqllen fields. The embedded SQL type is obtained by a bitwise AND of sqltype with DT_TYPES (sqltype & DT_TYPES).

For information about embedded SQL data types, see "Embedded SQL data types" on page 518.

Database field type	Embedded SQL type returned	Length (in bytes) returned on describe
BIGINT	DT_BIGINT	8
BINARY(n)	DT_BINARY	n
BIT	DT_BIT	1
CHAR(n)	DT_FIXCHAR	n
DATE	DT_DATE	length of longest formatted string

Database field type	Embedded SQL type returned	Length (in bytes) returned on describe
DECIMAL(p,s)	DT_DECIMAL	high byte of length field in SQLDA set to p, and low byte set to s
DOUBLE	DT_DOUBLE	8
FLOAT	DT_FLOAT	4
INT	DT_INT	4
LONG BINARY	DT_LONGBINARY	32767
LONG NVARCHAR	DT_LONGNVARCHAR[1]	32767
LONG VARCHAR	DT_LONGVARCHAR	32767
NCHAR(n)	DT_NFIXCHAR[1]	n times maximum character length in client's NCHAR character set
NVARCHAR(n)	DT_NVARCHAR[1]	n times maximum character length in client's NCHAR character set
REAL	DT_FLOAT	4
SMALLINT	DT_SMALLINT	2
TIME	DT_TIME	length of longest formatted string
TIMESTAMP	DT_TIMESTAMP	length of longest formatted string
TINYINT	DT_TINYINT	1
UNSIGNED BIGINT	DT_UNSBIGINT	8
UNSIGNED INT	DT_UNSINT	4
UNSIGNED SMALLINT	DT_UNSSMALLINT	2
VARCHAR(n)	DT_VARCHAR	n

[1] In embedded SQL, NCHAR, NVARCHAR, and LONG NVARCHAR are described as either DT_FIXCHAR, DT_VARCHAR, and DT_LONGVARCHAR, respectively, by default. If the db_change_nchar_charset function has been called, the types are described as DT_NFIXCHAR,

DT_NVARCHAR, and DT_LONGNVARCHAR, respectively. See "db_change_nchar_charset function" on page 574.

Sending values

The following table indicates how you specify lengths of values when you supply data to the database server in the SQLDA.

Only the data types displayed in the table are allowed in this case. The DT_DATE, DT_TIME, and DT_TIMESTAMP types are treated the same as DT_STRING when supplying information to the database; the value must be a null-terminated character string in an appropriate date format.

Embedded SQL data type	Program action to set the length
DT_BIGINT	No action required.
DT_BINARY(n)	Length taken from field in BINARY structure.
DT_BIT	No action required.
DT_DATE	Length determined by terminating \0.
DT_DOUBLE	No action required.
DT_FIXCHAR(n)	Length field in SQLDA determines length of string.
DT_FLOAT	No action required.
DT_INT	No action required.
DT_LONGBINARY	Length field ignored. See "Sending LONG data" on page 560.
DT_LONGNVARCHAR	Length field ignored. See "Sending LONG data" on page 560.
DT_LONGVARCHAR	Length field ignored. See "Sending LONG data" on page 560.
DT_NFIXCHAR(n)	Length field in SQLDA determines length of string.
DT_NSTRING	Length determined by terminating \0. If the ansi_blanks option is On and the database is blank-padded, then the length field in the SQLDA must be set to the length of the buffer containing the value (at least the length of the value plus space for the terminating null character).
DT_NVARCHAR	Length taken from field in NVARCHAR structure.

Embedded SQL data type	Program action to set the length
DT_SMALLINT	No action required.
DT_STRING	Length determined by terminating \0. If the ansi_blanks option is On and the database is blank-padded, then the length field in the SQLDA must be set to the length of the buffer containing the value (at least the length of the value plus space for the terminating null character).
DT_TIME	Length determined by terminating \0.
DT_TIMESTAMP	Length determined by terminating \0.
DT_TIMESTAMP_STRUCT	No action required.
DT_UNSBIGINT	No action required.
DT_UNSINT	No action required.
DT_UNSSMALLINT	No action required.
DT_VARCHAR(n)	Length taken from field in VARCHAR structure.
DT_VARIABLE	Length determined by terminating \0.

Retrieving values

The following table indicates the values of the length field when you retrieve data from the database using a SQLDA. The sqllen field is never modified when you retrieve data.

Only the interface data types displayed in the table are allowed in this case. The DT_DATE, DT_TIME, and DT_TIMESTAMP data types are treated the same as DT_STRING when you retrieve information from the database. The value is formatted as a character string in the current date format.

Embedded SQL data type	What the program must set length field to when receiving	How the database returns length information after fetching a value
DT_BIGINT	No action required.	No action required.
DT_BINARY(n)	Maximum length of BINARY structure (n+2). The maximum value for n is 32765.	len field of BINARY structure set to actual length in bytes.
DT_BIT	No action required.	No action required.

Embedded SQL data type	What the program must set length field to when receiving	How the database returns length information after fetching a value
DT_DATE	Length of buffer.	\0 at end of string.
DT_DOUBLE	No action required.	No action required.
DT_FIXCHAR(n)	Length of buffer, in bytes. The maximum value for n is 32767.	Padded with blanks to length of buffer.
DT_FLOAT	No action required.	No action required.
DT_INT	No action required.	No action required.
DT_LONGBINARY	Length field ignored. See "Retrieving LONG data" on page 559.	Length field ignored. See "Retrieving LONG data" on page 559.
DT_LONGNVARCHAR	Length field ignored. See "Retrieving LONG data" on page 559.	Length field ignored. See "Retrieving LONG data" on page 559.
DT_LONGVARCHAR	Length field ignored. See "Retrieving LONG data" on page 559.	Length field ignored. See "Retrieving LONG data" on page 559.
DT_NFIXCHAR(n)	Length of buffer, in bytes. The maximum value for n is 32767.	Padded with blanks to length of buffer.
DT_NSTRING	Length of buffer.	\0 at end of string.
DT_NVARCHAR(n)	Maximum length of NVARCHAR structure (n+2). The maximum value for n is 32765.	len field of NVARCHAR structure set to actual length in bytes of string.
DT_SMALLINT	No action required.	No action required.
DT_STRING	Length of buffer.	\0 at end of string.
DT_TIME	Length of buffer.	\0 at end of string.
DT_TIMESTAMP	Length of buffer.	\0 at end of string.
DT_TIMESTAMP_ STRUCT	No action required.	No action required.
DT_UNSBIGINT	No action required.	No action required.
DT_UNSINT	No action required.	No action required.

Embedded SQL data type	What the program must set length field to when receiving	How the database returns length information after fetching a value
DT_UNSSMALLINT	No action required.	No action required.
DT_VARCHAR(n)	Maximum length of VARCHAR structure (n+2). The maximum value for n is 32765.	len field of VARCHAR structure set to actual length in bytes of string.

Fetching data

Fetching data in embedded SQL is done using the SELECT statement. There are two cases:

- **The SELECT statement returns at most one row** Use an INTO clause to assign the returned values directly to host variables. See "SELECT statements that return at most one row" on page 550.
- **The SELECT statement may return multiple rows** Use cursors to manage the rows of the result set. See "Using cursors in embedded SQL" on page 551.

SELECT statements that return at most one row

A single row query retrieves at most one row from the database. A single-row query SELECT statement has an INTO clause following the select list and before the FROM clause. The INTO clause contains a list of host variables to receive the value for each select list item. There must be the same number of host variables as there are select list items. The host variables may be accompanied by indicator variables to indicate NULL results.

When the SELECT statement is executed, the database server retrieves the results and places them in the host variables. If the query results contain more than one row, the database server returns an error.

If the query results in no rows being selected, a `Row Not Found` warning is returned. Errors and warnings are returned in the SQLCA structure. See "The SQL Communication Area (SQLCA)" on page 531.

Example

The following code fragment returns 1 if a row from the Employees table is fetched successfully, 0 if the row doesn't exist, and -1 if an error occurs.

```
EXEC SQL BEGIN DECLARE SECTION;
long   ID;
char   name[41];
char   Sex;
char   birthdate[15];
short int ind_birthdate;
EXEC SQL END DECLARE SECTION;
...
int find_employee( long Employees )
{
  ID = Employees;
  EXEC SQL SELECT GivenName ||
    ' ' || Surname, Sex, BirthDate
    INTO :name, :Sex,
      :birthdate:ind_birthdate
    FROM Employees
    WHERE EmployeeID = :ID;
  if( SQLCODE == SQLE_NOTFOUND )
  {
    return( 0 ); /* Employees not found */
  }
  else if( SQLCODE < 0 )
  {
    return( -1 ); /* error */
  }
  else
```



```
    {
      return( 1 ); /* found */
    }
  }
```

Using cursors in embedded SQL

A cursor is used to retrieve rows from a query that has multiple rows in its result set. A **cursor** is a handle or an identifier for the SQL query and a position within the result set.

For an introduction to cursors, see "Working with cursors" on page 30.

To manage a cursor in embedded SQL

1. Declare a cursor for a particular SELECT statement, using the DECLARE statement.
2. Open the cursor using the OPEN statement.
3. Retrieve results one row at a time from the cursor using the FETCH statement.
4. Fetch rows until the `Row Not Found` warning is returned.

   Errors and warnings are returned in the SQLCA structure. See "The SQL Communication Area (SQLCA)" on page 531.
5. Close the cursor, using the CLOSE statement.

By default, cursors are automatically closed at the end of a transaction (on COMMIT or ROLLBACK). Cursors that are opened with a WITH HOLD clause are kept open for subsequent transactions until they are explicitly closed.

The following is a simple example of cursor usage:

```
void print_employees( void )
{
  EXEC SQL BEGIN DECLARE SECTION;
  char name[50];
  char Sex;
  char birthdate[15];
  short int ind_birthdate;
  EXEC SQL END DECLARE SECTION;
  EXEC SQL DECLARE C1 CURSOR FOR
    SELECT GivenName || ' ' || Surname,
      Sex, BirthDate
    FROM Employees;
  EXEC SQL OPEN C1;
  for( ;; )
  {
    EXEC SQL FETCH C1 INTO :name, :Sex,
        :birthdate:ind_birthdate;
    if( SQLCODE == SQLE_NOTFOUND )
    {
      break;
    }
    else if( SQLCODE < 0 )
    {
      break;
    }
```

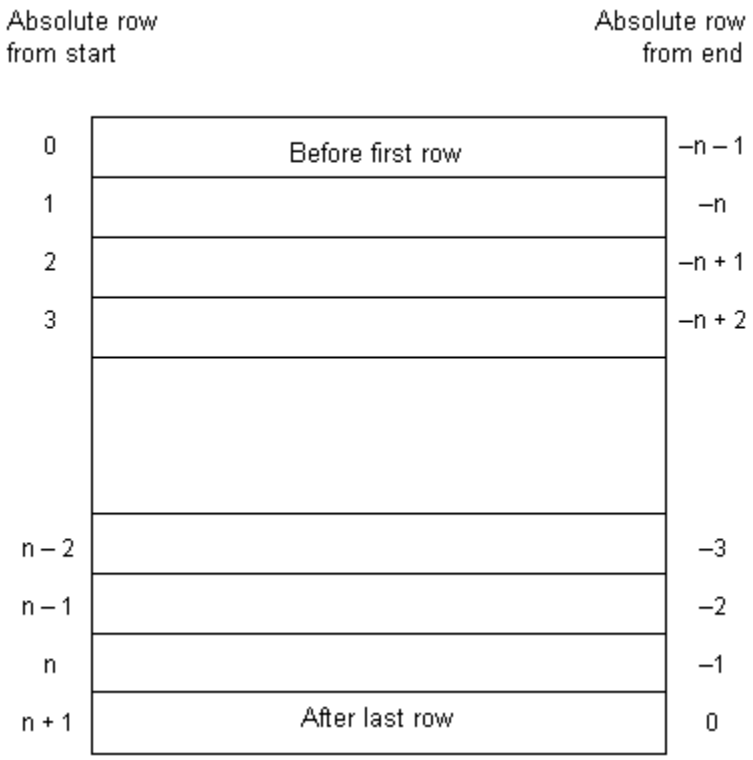
```
        if( ind_birthdate < 0 )
        {
          strcpy( birthdate, "UNKNOWN" );
        }
        printf( "Name: %s Sex: %c Birthdate:
             %s.n",name, Sex, birthdate );
    }
    EXEC SQL CLOSE C1;
}
```

For complete examples using cursors, see "Static cursor sample" on page 516 and "Dynamic cursor sample" on page 516.

Cursor positioning

A cursor is positioned in one of three places:

- On a row
- Before the first row
- After the last row



When a cursor is opened, it is positioned before the first row. The cursor position can be moved using the FETCH statement. It can be positioned to an absolute position either from the start or from the end of the query results. It can also be moved relative to the current cursor position. See "FETCH statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].

There are special **positioned** versions of the UPDATE and DELETE statements that can be used to update or delete the row at the current position of the cursor. If the cursor is positioned before the first row or after the last row, an error is returned indicating that there is no corresponding row in the cursor.

The PUT statement can be used to insert a row into a cursor. See "PUT statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

Cursor positioning problems

Inserts and some updates to DYNAMIC SCROLL cursors can cause problems with cursor positioning. The database server does not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row does not appear at all until the cursor is closed and opened again.

With SQL Anywhere, this occurs if a temporary table had to be created to open the cursor.

For a description, see "Use work tables in query processing (use All-rows optimization goal)" [*SQL Anywhere Server - SQL Usage*].

The UPDATE statement can cause a row to move in the cursor. This happens if the cursor has an ORDER BY clause that uses an existing index (a temporary table is not created).

Fetching more than one row at a time

The FETCH statement can be modified to fetch more than one row at a time, which may improve performance. This is called a **wide fetch** or an **array fetch**.

SQL Anywhere also supports wide puts and inserts. See "PUT statement [ESQL]" [*SQL Anywhere Server - SQL Reference*] and "EXECUTE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

To use wide fetches in embedded SQL, include the fetch statement in your code as follows:

```
EXEC SQL FETCH ... ARRAY nnn
```

where ARRAY *nnn* is the last item of the FETCH statement. The fetch count *nnn* can be a host variable. The number of variables in the SQLDA must be the product of *nnn* and the number of columns per row. The first row is placed in SQLDA variables 0 to (columns per row) - 1, and so on.

Each column must be of the same type in each row of the SQLDA, or a SQLDA_INCONSISTENT error is returned.

The server returns in SQLCOUNT the number of records that were fetched, which is always greater than zero unless there is an error or warning. On a wide fetch, a SQLCOUNT of one with no error condition indicates that one valid row has been fetched.

Example

The following example code illustrates the use of wide fetches. You can also find this code in *samples-dir\SQLAnywhere\esqlwidefetch\widefetch.sqc*.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include "sqldef.h"
EXEC SQL INCLUDE SQLCA;

EXEC SQL WHENEVER SQLERROR { PrintSQLError();
        goto err; };

static void PrintSQLError()
{
  char buffer[200];

  printf( "SQL error %d -- %s\n",
     SQLCODE,
     sqlerror_message( &sqlca,
              buffer,
              sizeof( buffer ) ) );
}
static SQLDA * PrepareSQLDA(
  a_sql_statement_number stat0,
  unsigned  width,
  unsigned  *cols_per_row )

/* Allocate a SQLDA to be used for fetching from
   the statement identified by "stat0". "width"
   rows are retrieved on each FETCH request.
   The number of columns per row is assigned to
   "cols_per_row". */
{
  int           num_cols;
  unsigned      row, col, offset;
  SQLDA *       sqlda;
  EXEC SQL BEGIN DECLARE SECTION;
  a_sql_statement_number  stat;
  EXEC SQL END DECLARE SECTION;
  stat = stat0;
  sqlda = alloc_sqlda( 100 );
  if( sqlda == NULL ) return( NULL );
  EXEC SQL DESCRIBE :stat INTO sqlda;
  *cols_per_row = num_cols = sqlda->sqld;
  if( num_cols * width > sqlda->sqln )
  {
    free_sqlda( sqlda );
    sqlda = alloc_sqlda( num_cols * width );
    if( sqlda == NULL ) return( NULL );
    EXEC SQL DESCRIBE :stat INTO sqlda;
  }
  // copy first row in SQLDA setup by describe
  // to following (wide) rows
  sqlda->sqld = num_cols * width;
  offset = num_cols;
  for( row = 1; row < width; row++ )
  {
    for( col = 0;
      col < num_cols;
      col++, offset++ )
    {
      sqlda->sqlvar[offset].sqltype =
          sqlda->sqlvar[col].sqltype;
      sqlda->sqlvar[offset].sqllen =
          sqlda->sqlvar[col].sqllen;
      // optional: copy described column name
      memcpy( &sqlda->sqlvar[offset].sqlname,
              &sqlda->sqlvar[col].sqlname,
              sizeof( sqlda->sqlvar[0].sqlname ) );
    }
```



```
  }
  fill_s_sqlda( sqlda, 40 );
  return( sqlda );
err:
  return( NULL );
}
static void PrintFetchedRows(
  SQLDA * sqlda,
  unsigned cols_per_row )
{
  /* Print rows already wide fetched in the SQLDA */
  long      rows_fetched;
  int       row, col, offset;

  if( SQLCOUNT == 0 )
  {
    rows_fetched = 1;
  }
  else
  {
    rows_fetched = SQLCOUNT;
  }
  printf( "Fetched %d Rows:\n", rows_fetched );
  for( row = 0; row < rows_fetched; row++ )
  {
    for( col = 0; col < cols_per_row; col++ )
    {
      offset = row * cols_per_row + col;
      printf( " \"%s\"",
        (char *)sqlda->sqlvar[offset].sqldata );
     }
     printf( "\n" );
  }
}
static int DoQuery(
  char * query_str0,
  unsigned fetch_width0 )
{
  /* Wide Fetch "query_str0" select statement
   * using a width of "fetch_width0" rows" */
  SQLDA *                 sqlda;
  unsigned                cols_per_row;
  EXEC SQL BEGIN DECLARE SECTION;
  a_sql_statement_number  stat;
  char *                  query_str;
  unsigned                fetch_width;
  EXEC SQL END DECLARE SECTION;

  query_str = query_str0;
  fetch_width = fetch_width0;

  EXEC SQL PREPARE :stat FROM :query_str;
  EXEC SQL DECLARE QCURSOR CURSOR FOR :stat
       FOR READ ONLY;
  EXEC SQL OPEN QCURSOR;
  sqlda = PrepareSQLDA( stat,
    fetch_width,
    &cols_per_row );
  if( sqlda == NULL )
  {
    printf( "Error allocating SQLDA\n" );
    return( SQLE_NO_MEMORY );
  }
  for( ;; )
```

```
    {
      EXEC SQL FETCH QCURSOR INTO DESCRIPTOR sqlda
        ARRAY :fetch_width;
      if( SQLCODE != SQLE_NOERROR ) break;
      PrintFetchedRows( sqlda, cols_per_row );
    }
    EXEC SQL CLOSE QCURSOR;
    EXEC SQL DROP STATEMENT :stat;
    free_filled_sqlda( sqlda );
  err:
    return( SQLCODE );
  }
  void main( int argc, char *argv[] )
  {
    /* Optional first argument is a select statement,
     * optional second argument is the fetch width */
    char *query_str =
      "select GivenName, Surname from Employees";
    unsigned fetch_width = 10;

    if( argc > 1 )
    {
      query_str = argv[1];
      if( argc > 2 )
      {
        fetch_width = atoi( argv[2] );
        if( fetch_width < 2 )
        {
          fetch_width = 2;
        }
      }
    }
    db_init( &sqlca );
    EXEC SQL CONNECT "DBA" IDENTIFIED BY "sql";

    DoQuery( query_str, fetch_width );

    EXEC SQL DISCONNECT;
  err:
    db_fini( &sqlca );
  }
```

Notes on using wide fetches

- In the function PrepareSQLDA, the SQLDA memory is allocated using the alloc_sqlda function. This allows space for indicator variables, rather than using the alloc_sqlda_noind function.
- If the number of rows fetched is fewer than the number requested, but is not zero (at the end of the cursor for example), the SQLDA items corresponding to the rows that were not fetched are returned as NULL by setting the indicator value. If no indicator variables are present, an error is generated (SQLE_NO_INDICATOR: no indicator variable for NULL result).
- If a row being fetched has been updated, generating a SQLE_ROW_UPDATED_WARNING warning, the fetch stops on the row that caused the warning. The values for all rows processed to that point (including the row that caused the warning) are returned. SQLCOUNT contains the number of rows that were fetched, including the row that caused the warning. All remaining SQLDA items are marked as NULL.
- If a row being fetched has been deleted or is locked, generating a SQLE_NO_CURRENT_ROW or SQLE_LOCKED error, SQLCOUNT contains the number of rows that were read prior to the error. This

does not include the row that caused the error. The SQLDA does not contain values for any of the rows since SQLDA values are not returned on errors. The SQLCOUNT value can be used to reposition the cursor, if necessary, to read the rows.

Sending and retrieving long values

The method for sending and retrieving LONG VARCHAR, LONG NVARCHAR, and LONG BINARY values in embedded SQL applications is different from that for other data types. The standard SQLDA fields are limited to 32767 bytes of data as the fields holding the length information (sqldata, sqllen, sqlind) are 16-bit values. Changing these values to 32-bit values would break existing applications.

The method of describing LONG VARCHAR, LONG NVARCHAR, and LONG BINARY values is the same as for other data types.

For information about how to retrieve and send values, see "Retrieving LONG data" on page 559, and "Sending LONG data" on page 560.

Static SQL structures

Separate fields are used to hold the allocated, stored, and untruncated lengths of LONG BINARY, LONG VARCHAR, and LONG NVARCHAR data types. The static SQL data types are defined in *sqlca.h* as follows:

```
#define DECL_LONGVARCHAR( size )         \
  struct { a_sql_uint32    array_len;    \
           a_sql_uint32    stored_len;   \
           a_sql_uint32    untrunc_len;  \
           char            array[size+1];\
         }
#define DECL_LONGNVARCHAR( size )        \
  struct { a_sql_uint32    array_len;    \
           a_sql_uint32    stored_len;   \
           a_sql_uint32    untrunc_len;  \
           char            array[size+1];\
         }
#define DECL_LONGBINARY( size )          \
  struct { a_sql_uint32    array_len;    \
           a_sql_uint32    stored_len;   \
           a_sql_uint32    untrunc_len;  \
           char            array[size];  \
         }
```

Dynamic SQL structures

For dynamic SQL, set the sqltype field to DT_LONGVARCHAR, DT_LONGNVARCHAR, or DT_LONGBINARY as appropriate. The associated LONGVARCHAR, LONGNVARCHAR, and LONGBINARY structures are as follows:

```
typedef struct LONGVARCHAR {
    a_sql_uint32    array_len;
    a_sql_uint32    stored_len;
    a_sql_uint32    untrunc_len;
    char            array[1];
} LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
```

Structure member definitions

For both static and dynamic SQL structures, the structure members are defined as follows:

- **array_len** (Sending and retrieving.) The number of bytes allocated for the array part of the structure.

- **stored_len** (Sending and retrieving.) The number of bytes stored in the array. Always less than or equal to array_len and untrunc_len.
- **untrunc_len** (Retrieving only.) The number of bytes that would be stored in the array if the value was not truncated. Always greater than or equal to stored_len. If truncation occurs, this value is larger than array_len.

Retrieving LONG data

This section describes how to retrieve LONG values from the database. For background information, see "Sending and retrieving long values" on page 558.

The procedures are different depending on whether you are using static or dynamic SQL.

To receive a LONG VARCHAR, LONG NVARCHAR, or LONG BINARY value (static SQL)

1. Declare a host variable of type DECL_LONGVARCHAR, DECL_LONGNVARCHAR, or DECL_LONGBINARY, as appropriate. The array_len member is filled in automatically.
2. Retrieve the data using FETCH, GET DATA, or EXECUTE INTO. SQL Anywhere sets the following information:
   - **indicator variable** Negative if the value is NULL, 0 if there is no truncation, otherwise the positive untruncated length in bytes up to a maximum of 32767.

     For more information, see "Indicator variables" on page 527.
   - **stored_len** The number of bytes stored in the array. Always less than or equal to array_len and untrunc_len.
   - **untrunc_len** The number of bytes that would be stored in the array if the value was not truncated. Always greater than or equal to stored_len. If truncation occurs, this value is larger than array_len.

To receive a value into a LONGVARCHAR, LONGNVARCHAR, or LONGBINARY structure (dynamic SQL)

1. Set the sqltype field to DT_LONGVARCHAR, DT_LONGNVARCHAR, or DT_LONGBINARY as appropriate.
2. Set the sqldata field to point to the LONGVARCHAR, LONGNVARCHAR, or LONGBINARY host variable structure.

   You can use the `LONGVARCHARSIZE( n )`, `LONGNVARCHARSIZE( n )`, or `LONGBINARYSIZE( n )` macro to determine the total number of bytes to allocate to hold `n` bytes of data in the array field.
3. Set the array_len field of the host variable structure to the number of bytes allocated for the array field.
4. Retrieve the data using FETCH, GET DATA, or EXECUTE INTO. SQL Anywhere sets the following information:
   - *** sqlind** This sqlda field is negative if the value is NULL, 0 if there is no truncation, and is the positive untruncated length in bytes up to a maximum of 32767.

- **stored_len** The number of bytes stored in the array. Always less than or equal to array_len and untrunc_len.
- **untrunc_len** The number of bytes that would be stored in the array if the value was not truncated. Always greater than or equal to stored_len. If truncation occurs, this value is larger than array_len.

The following code fragment illustrates the mechanics of retrieving LONG VARCHAR data using dynamic embedded SQL. It is not intended to be a practical application:

```
#define DATA_LEN 128000
void get_test_var()
{
  LONGVARCHAR *longptr;
  SQLDA       *sqlda;
  SQLVAR      *sqlvar;

  sqlda = alloc_sqlda( 1 );
  longptr = (LONGVARCHAR *)malloc(
              LONGVARCHARSIZE( DATA_LEN ) );
  if( sqlda == NULL || longptr == NULL )
  {
    fatal_error( "Allocation failed" );
  }

  // init longptr for receiving data
  longptr->array_len = DATA_LEN;

  // init sqlda for receiving data
  // (sqllen is unused with DT_LONG types)
  sqlda->sqld = 1; // using 1 sqlvar
  sqlvar = &sqlda->sqlvar[0];
  sqlvar->sqltype = DT_LONGVARCHAR;
  sqlvar->sqldata = longptr;
  printf( "fetching test_var\n" );
  EXEC SQL PREPARE select_stmt FROM 'SELECT test_var';
  EXEC SQL EXECUTE select_stmt INTO DESCRIPTOR sqlda;
  EXEC SQL DROP STATEMENT select_stmt;
  printf( "stored_len: %d, untrunc_len: %d, "
          "1st char: %c, last char: %c\n",
        longptr->stored_len,
        longptr->untrunc_len,
        longptr->array[0],
        longptr->array[DATA_LEN-1] );
  free_sqlda( sqlda );
  free( longptr );
}
```

Sending LONG data

This section describes how to send LONG values to the database from embedded SQL applications. For background information, see “Sending and retrieving long values” on page 558.

The procedures are different depending on whether you are using static or dynamic SQL.

To send a LONG value (static SQL)

1. Declare a host variable of type DECL_LONGVARCHAR, DECL_LONGNVARCHAR, or DECL_LONGBINARY, as appropriate.

2. If you are sending NULL, set the indicator variable to a negative value.

   For more information, see "Indicator variables" on page 527.

3. Set the stored_len field of the host variable structure to the number of bytes of data in the array field.
4. Send the data by opening the cursor or executing the statement.

The following code fragment illustrates the mechanics of sending a LONG VARCHAR using static embedded SQL. It is not intended to be a practical application.

```
#define DATA_LEN 12800
EXEC SQL BEGIN DECLARE SECTION;
// SQLPP initializes longdata.array_len
DECL_LONGVARCHAR(128000) longdata;
EXEC SQL END DECLARE SECTION;

void set_test_var()
{
  // init longdata for sending data
  memset( longdata.array, 'a', DATA_LEN );
  longdata.stored_len = DATA_LEN;

  printf( "Setting test_var to %d a's\n", DATA_LEN );
  EXEC SQL SET test_var = :longdata;
}
```

To send a LONG value (dynamic SQL)

1. Set the sqltype field to DT_LONGVARCHAR, DT_LONGNVARCHAR, or DT_LONGBINARY, as appropriate.
2. If you are sending NULL, set *** sqlind** to a negative value.
3. If you are not sending NULL, set the sqldata field to point to the LONGVARCHAR, LONGNVARCHAR, or LONGBINARY host variable structure.

   You can use the `LONGVARCHARSIZE( n )`, `LONGNVARCHARSIZE( n )`, or `LONGBINARYSIZE( n )` macros to determine the total number of bytes to allocate to hold `n` bytes of data in the array field.

4. Set the array_len field of the host variable structure to the number of bytes allocated for the array field.
5. Set the stored_len field of the host variable structure to the number of bytes of data in the array field. This must not be more than array_len.
6. Send the data by opening the cursor or executing the statement.

Using simple stored procedures

You can create and call stored procedures in embedded SQL.

You can embed a CREATE PROCEDURE just like any other data definition statement, such as CREATE TABLE. You can also embed a CALL statement to execute a stored procedure. The following code fragment illustrates both creating and executing a stored procedure in embedded SQL:

```
EXEC SQL CREATE PROCEDURE pettycash(
 IN Amount DECIMAL(10,2) )
BEGIN
 UPDATE account
 SET balance = balance - Amount
 WHERE name = 'bank';

 UPDATE account
 SET balance = balance + Amount
 WHERE name = 'pettycash expense';
END;
EXEC SQL CALL pettycash( 10.72 );
```

If you want to pass host variable values to a stored procedure or to retrieve the output variables, you prepare and execute a CALL statement. The following code fragment illustrates the use of host variables. Both the USING and INTO clauses are used on the EXECUTE statement.

```
EXEC SQL BEGIN DECLARE SECTION;
double  hv_expense;
double  hv_balance;
EXEC SQL END DECLARE SECTION;

// Code here
EXEC SQL CREATE PROCEDURE pettycash(
   IN expense  DECIMAL(10,2),
   OUT endbalance DECIMAL(10,2) )
BEGIN
  UPDATE account
  SET balance = balance - expense
  WHERE name = 'bank';
  UPDATE account
  SET balance = balance + expense
  WHERE name = 'pettycash expense';

  SET endbalance = ( SELECT balance FROM account
         WHERE name = 'bank' );
END;

EXEC SQL PREPARE S1 FROM 'CALL pettycash( ?, ? )';
EXEC SQL EXECUTE S1 USING :hv_expense INTO :hv_balance;
```

For more information, see "EXECUTE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*], and "PREPARE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

Stored procedures with result sets

Database procedures can also contain SELECT statements. The procedure is declared using a RESULT clause to specify the number, name, and types of the columns in the result set. Result set columns are different

from output parameters. For procedures with result sets, the CALL statement can be used in place of a SELECT statement in the cursor declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
 char hv_name[100];
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE PROCEDURE female_employees()
  RESULT( name char(50) )
BEGIN
  SELECT GivenName || Surname FROM Employees
  WHERE Sex = 'f';
END;

EXEC SQL PREPARE S1 FROM 'CALL female_employees()';

EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
for(;;)
{
  EXEC SQL FETCH C1 INTO :hv_name;
  if( SQLCODE != SQLE_NOERROR ) break;
  printf( "%s\\n", hv_name );
}
EXEC SQL CLOSE C1;
```

In this example, the procedure has been invoked with an OPEN statement rather than an EXECUTE statement. The OPEN statement causes the procedure to execute until it reaches a SELECT statement. At this point, C1 is a cursor for the SELECT statement within the database procedure. You can use all forms of the FETCH statement (backward and forward scrolling) until you are finished with it. The CLOSE statement stops execution of the procedure.

If there had been another statement following the SELECT in the procedure, it would not have been executed. To execute statements following a SELECT, use the RESUME cursor-name statement. The RESUME statement either returns the warning SQLE_PROCEDURE_COMPLETE or it returns SQLE_NOERROR indicating that there is another cursor. The example illustrates a two-select procedure:

```
EXEC SQL CREATE PROCEDURE people()
  RESULT( name char(50) )
BEGIN
 SELECT GivenName || Surname
 FROM Employees;

 SELECT GivenName || Surname
 FROM Customers;
END;

EXEC SQL PREPARE S1 FROM 'CALL people()';
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
while( SQLCODE == SQLE_NOERROR )
{
  for(;;)
  {
    EXEC SQL FETCH C1 INTO :hv_name;
    if( SQLCODE != SQLE_NOERROR ) break;
    printf( "%s\\n", hv_name );
  }
  EXEC SQL RESUME C1;
}
EXEC SQL CLOSE C1;
```

Dynamic cursors for CALL statements

These examples have used static cursors. Full dynamic cursors can also be used for the CALL statement.

For a description of dynamic cursors, see "Dynamic SELECT statement" on page 539.

The DESCRIBE statement works fully for procedure calls. A DESCRIBE OUTPUT produces a SQLDA that has a description for each of the result set columns.

If the procedure does not have a result set, the SQLDA has a description for each INOUT or OUT parameter for the procedure. A DESCRIBE INPUT statement produces a SQLDA having a description for each IN or INOUT parameter for the procedure.

DESCRIBE ALL

DESCRIBE ALL describes IN, INOUT, OUT, and RESULT set parameters. DESCRIBE ALL uses the indicator variables in the SQLDA to provide additional information.

The DT_PROCEDURE_IN and DT_PROCEDURE_OUT bits are set in the indicator variable when a CALL statement is described. DT_PROCEDURE_IN indicates an IN or INOUT parameter and DT_PROCEDURE_OUT indicates an INOUT or OUT parameter. Procedure RESULT columns have both bits clear.

After a describe OUTPUT, these bits can be used to distinguish between statements that have result sets (need to use OPEN, FETCH, RESUME, CLOSE) and statements that do not (need to use EXECUTE).

For a complete description, see "DESCRIBE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

Multiple result sets

If you have a procedure that returns multiple result sets, you must re-describe after each RESUME statement if the result sets change shapes.

You need to describe the cursor, not the statement, to re-describe the current position of the cursor.

Embedded SQL programming techniques

This section contains a set of tips for developers of embedded SQL programs.

Implementing request management

The default behavior of the interface DLL is for applications to wait for completion of each database request before carrying out other functions. This behavior can be changed using request management functions. For example, when using Interactive SQL, the operating system is still active while Interactive SQL is waiting for a response from the database and Interactive SQL carries out some tasks in that time.

You can achieve application activity while a database request is in progress by providing a callback function. In this callback function, you must not do another database request except db_cancel_request. You can use the db_is_working function in your message handlers to determine if you have a database request in progress.

The db_register_a_callback function is used to register your application callback functions.

See also

- "db_register_a_callback function" on page 580
- "db_cancel_request function" on page 573
- "db_is_working function" on page 577

Backup functions

The db_backup function provides support for online backup in embedded SQL applications. The backup utility makes use of this function. You should only need to write a program to use this function if your backup requirements are not satisfied by the SQL Anywhere backup utility.

BACKUP statement is recommended
Although this function provides one way to add backup features to an application, the recommended way to accomplish this task is to use the BACKUP statement. See "BACKUP statement" [*SQL Anywhere Server - SQL Reference*].

You can also access the backup utility directly using the Database Tools DBBackup function. See "DBBackup function" on page 821.

See also

- "db_backup function" on page 569

SQL preprocessor

The SQL preprocessor processes a C or C++ program containing embedded SQL before the compiler is run.

Syntax

sqlpp [*options*] *input-file* [*output-file*]

Option	Description
-d	Generate code that reduces data space size. Data structures are reused and initialized at execution time before use. This increases code size.
-e *level*	Flag as an error any static embedded SQL that is not part of a specified standard. The *level* value indicates the standard to use. For example, `sqlpp -e c03 ...` flags any syntax that is not part of the core SQL/2003 standard. The supported *level* values are: • **c03** Flag syntax that is not core SQL/2003 syntax • **p03** Flag syntax that is not full SQL/2003 syntax • **c99** Flag syntax that is not core SQL/1999 syntax • **p99** Flag syntax that is not full SQL/1999 syntax • **e92** Flag syntax that is not entry-level SQL/1992 syntax • **i92** Flag syntax that is not intermediate-level SQL/1992 syntax • **f92** Flag syntax that is not full-SQL/1992 syntax • **t** Flag non-standard host variable types • **u** Flag syntax that is not supported by UltraLite For compatibility with previous SQL Anywhere versions, you can also specify e, i, and f, which correspond to e92, i92, and f92, respectively.
-h *width*	Limit the maximum length of lines output by sqlpp to *width*. The continuation character is a backslash (\) and the minimum value of *width* is 10.
-k	Notify the preprocessor that the program to be compiled includes a user declaration of SQLCODE. The definition must be of type LONG, but does not need to be in a declaration section.
-n	Generate line number information in the C file. This consists of *#line* directives in the appropriate places in the generated C code. If the compiler that you are using supports the *#line* directive, this option makes the compiler report errors on line numbers in the SQC file (the one with the embedded SQL) as opposed to reporting errors on line numbers in the C file generated by the SQL preprocessor. Also, the *#line* directives are used indirectly by the source level debugger so that you can debug while viewing the SQC source file.

Option	Description
-o *operating-system*	Specify the target operating system. The supported operating systems are: • **WINDOWS** Microsoft Windows • **UNIX** Use this option if you are creating a 32-bit Unix application. • **UNIX64** Use this option if you are creating a 64-bit Unix application.
-q	Quiet mode—do not print messages.
-r-	Generate non-reentrant code. For more information about reentrant code, see "SQLCA management for multi-threaded or reentrant code" on page 533.
-s *len*	Set the maximum size string that the preprocessor puts into the C file. Strings longer than this value are initialized using a list of characters (*'a','b','c'*, and so on). Most C compilers have a limit on the size of string literal they can handle. This option is used to set that upper limit. The default value is 500.
-u	Generate code for UltraLite. For more information, see "Embedded SQL API reference" [*UltraLite - C and C++ Programming*].
-w *level*	Flag as a warning any static embedded SQL that is not part of a specified standard. The *level* value indicates the standard to use. For example, `sqlpp -w c03 ...` flags any SQL syntax that is not part of the core SQL/2003 syntax. The supported *level* values are: • **c03** Flag syntax that is not core SQL/2003 syntax • **p03** Flag syntax that is not full SQL/2003 syntax • **c99** Flag syntax that is not core SQL/1999 syntax • **p99** Flag syntax that is not full SQL/1999 syntax • **e92** Flag syntax that is not entry-level SQL/1992 syntax • **i92** Flag syntax that is not intermediate-level SQL/1992 syntax • **f92** Flag syntax that is not full-SQL/1992 syntax • **t** Flag non-standard host variable types • **u** Flag syntax that is not supported by UltraLite For compatibility with previous SQL Anywhere versions, you can also specify e, i, and f, which correspond to e92, i92, and f92, respectively.
-x	Change multibyte strings to escape sequences so that they can pass through compilers.

Option	Description
-z *cs*	Specify the collation sequence. For a list of recommended collation sequences, enter dbinit -l at a command prompt. The collation sequence is used to help the preprocessor understand the characters used in the source code of the program, for example, in identifying alphabetic characters suitable for use in identifiers. If -z is not specified, the preprocessor attempts to determine a reasonable collation to use based on the operating system and the SALANG and SACHARSET environment variables. See "SACHARSET environment variable" [*SQL Anywhere Server - Database Administration*], and "SALANG environment variable" [*SQL Anywhere Server - Database Administration*].
input-file	A C or C++ program containing embedded SQL to be processed.
output-file	The C language source file created by the SQL preprocessor.

Description

The SQL preprocessor translates the SQL statements in the *input-file* into C language source that is put into the *output-file*. The normal extension for source programs with embedded SQL is *.sqc*. The default output file name is the *input-file* with an extension of *.c*. If *input-file* has a *.c* extension, the default output file name extension is *.cc*.

See also

- "Introduction to embedded SQL" on page 508
- "sql_flagger_error_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "sql_flagger_warning_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "SQLFLAGGER function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "sa_ansi_standard_packages system procedure" [*SQL Anywhere Server - SQL Reference*]
- "SQL preprocessor error messages" [*Error Messages*]

Library function reference

The SQL preprocessor generates calls to functions in the interface library or DLL. In addition to the calls generated by the SQL preprocessor, a set of library functions is provided to make database operations easier to perform. Prototypes for these functions are included by the EXEC SQL INCLUDE SQLCA statement.

This section contains a reference description of these various functions.

DLL entry points

The DLL entry points are the same except that the prototypes have a modifier appropriate for DLLs.

You can declare the entry points in a portable manner using **_esqlentry_**, which is defined in *sqlca.h*. It resolves to the value __stdcall.

alloc_sqlda function

Prototype

struct sqlda * **alloc_sqlda(** unsigned *numvar* **);**

Description

Allocates a SQLDA with descriptors for *numvar* variables. The sqln field of the SQLDA is initialized to *numvar*. Space is allocated for the indicator variables, the indicator pointers are set to point to this space, and the indicator value is initialized to zero. A null pointer is returned if memory cannot be allocated. It is recommended that you use this function instead of the alloc_sqlda_noind function.

alloc_sqlda_noind function

Prototype

struct sqlda * **alloc_sqlda_noind(** unsigned *numvar* **);**

Description

Allocates a SQLDA with descriptors for *numvar* variables. The sqln field of the SQLDA is initialized to *numvar*. Space is not allocated for indicator variables; the indicator pointers are set to the null pointer. A null pointer is returned if memory cannot be allocated.

db_backup function

Prototype

void **db_backup(**
SQLCA * *sqlca***,**
int *op***,**
*int file_num***,**
unsigned long *page_num***,**
struct sqlda * *sqlda***);**

Authorization

Must be connected as a user with DBA authority, REMOTE DBA authority (SQL Remote), or BACKUP authority.

Description

BACKUP statement is recommended
Although this function provides one way to add backup features to an application, the recommended way to accomplish this task is to use the BACKUP statement. See "BACKUP statement" [*SQL Anywhere Server - SQL Reference*].

The action performed depends on the value of the *op* parameter:

- **DB_BACKUP_START** Must be called before a backup can start. Only one backup can be running per database at one time against any given database server. Database checkpoints are disabled until the backup is complete (db_backup is called with an *op* value of DB_BACKUP_END). If the backup cannot start, the SQLCODE is SQLE_BACKUP_NOT_STARTED. Otherwise, the SQLCOUNT field of the *sqlca* is set to the database page size. Backups are processed one page at a time.

  The *file_num*, *page_num*, and *sqlda* parameters are ignored.

- **DB_BACKUP_OPEN_FILE** Open the database file specified by *file_num*, which allows pages of the specified file to be backed up using DB_BACKUP_READ_PAGE. Valid file numbers are 0 through DB_BACKUP_MAX_FILE for the root database files, and 0 through DB_BACKUP_TRANS_LOG_FILE for the transaction log file. If the specified file does not exist, the SQLCODE is SQLE_NOTFOUND. Otherwise, SQLCOUNT contains the number of pages in the file, SQLIOESTIMATE contains a 32-bit value (POSIX time_t) that identifies the time that the database file was created, and the operating system file name is in the *sqlerrmc* field of the SQLCA.

  The *page_num* and *sqlda* parameters are ignored.

- **DB_BACKUP_READ_PAGE** Read one page of the database file specified by *file_num*. The *page_num* should be a value from 0 to one less than the number of pages returned in SQLCOUNT by a successful call to db_backup with the DB_BACKUP_OPEN_FILE operation. Otherwise, SQLCODE is set to SQLE_NOTFOUND. The *sqlda* descriptor should be set up with one variable of type DT_BINARY or DT_LONG_BINARY pointing to a buffer. The buffer should be large enough to hold binary data of the size returned in the SQLCOUNT field on the call to db_backup with the DB_BACKUP_START operation.

  DT_BINARY data contains a two-byte length followed by the actual binary data, so the buffer must be two bytes longer than the page size.

  Application must save buffer
  This call makes a copy of the specified database page into the buffer, but it is up to the application to save the buffer on some backup media.

- **DB_BACKUP_READ_RENAME_LOG** This action is the same as DB_BACKUP_READ_PAGE, except that after the last page of the transaction log has been returned, the database server renames the transaction log and starts a new one.

If the database server is unable to rename the log at the current time (for example in version 7.0.x or earlier databases there may be incomplete transactions), the SQLE_BACKUP_CANNOT_RENAME_LOG_YET error is set. In this case, do not use the page returned, but instead reissue the request until you receive SQLE_NOERROR and then write the page. Continue reading the pages until you receive the SQLE_NOTFOUND condition.

The SQLE_BACKUP_CANNOT_RENAME_LOG_YET error may be returned multiple times and on multiple pages. In your retry loop, you should add a delay so as not to slow the server down with too many requests.

When you receive the SQLE_NOTFOUND condition, the transaction log has been backed up successfully and the file has been renamed. The name for the old transaction file is returned in the *sqlerrmc* field of the SQLCA.

You should check the sqlda->sqlvar[0].sqlind value after a db_backup call. If this value is greater than zero, the last log page has been written and the log file has been renamed. The new name is still in sqlca.sqlerrmc, but the SQLCODE value is SQLE_NOERROR.

You should not call db_backup again after this, except to close files and finish the backup. If you do, you get a second copy of your backed up log file and you receive SQLE_NOTFOUND.

- **DB_BACKUP_CLOSE_FILE** Must be called when processing of one file is complete to close the database file specified by *file_num*.

  The *page_num* and *sqlda* parameters are ignored.

- **DB_BACKUP_END** Must be called at the end of the backup. No other backup can start until this backup has ended. Checkpoints are enabled again.

  The *file_num*, *page_num* and *sqlda* parameters are ignored.

- **DB_BACKUP_PARALLEL_START** Starts a parallel backup. Like DB_BACKUP_START, only one backup can be running against a database at one time on any given database server. Database checkpoints are disabled until the backup is complete (until db_backup is called with an *op* value of DB_BACKUP_END). If the backup cannot start, you receive SQLE_BACKUP_NOT_STARTED. Otherwise, the SQLCOUNT field of the sqlca is set to the database page size.

  The *file_num* parameter instructs the database server to rename the transaction log and start a new one after the last page of the transaction log has been returned. If the value is non-zero then the transaction log is renamed or restarted. Otherwise, it is not renamed and restarted. This parameter eliminates the need for the DB_BACKUP_READ_RENAME_LOG operation, which is not allowed during a parallel backup operation.

  The *page_num* parameter informs the database server of the maximum size of the client's buffer, in database pages. On the server side, the parallel backup readers try to read sequential blocks of pages—this value lets the server know how large to allocate these blocks: passing a value of *N* lets the server know that the client is willing to accept at most *N* database pages at a time from the server. The server may return blocks of pages of less than size *N* if it is unable to allocate enough memory for blocks of *N* pages. If the client does not know the size of database pages until after the call to DB_BACKUP_PARALLEL_START, this value can be provided to the server with the DB_BACKUP_INFO operation. This value must be provided before the first call to retrieve backup pages (DB_BACKUP_PARALLEL_READ).

Note
If you are using db_backup to start a parallel backup, db_backup does not create writer threads. The caller of db_backup must receive the data and act as the writer.

- **DB_BACKUP_INFO** This parameter provides additional information to the database server about the parallel backup. The *file_num* parameter indicates the type of information being provided, and the *page_num* parameter provides the value. You can specify the following additional information with DB_BACKUP_INFO:
  - **DB_BACKUP_INFO_PAGES_IN_BLOCK** The *page_num* argument contains the maximum number of pages that should be sent back in one block.
  - **DB_BACKUP_INFO_CHKPT_LOG** This is the client-side equivalent to the WITH CHECKPOINT LOG option of the BACKUP statement. A *page_num* value of DB_BACKUP_CHKPT_COPY indicates COPY, while the value DB_BACKUP_CHKPT_NOCOPY indicates NO COPY. If this value is not provided it defaults to COPY.
- **DB_BACKUP_PARALLEL_READ** This operation reads a block of pages from the database server. Before this operation is invoked, all of the files that are to be backed up must have been opened using the DB_BACKUP_OPEN_FILE operation. DB_BACKUP_PARALLEL_READ ignores the *file_num* and *page_num* arguments.

  The sqlda descriptor should be set up with one variable of type DT_LONGBINARY pointing to a buffer. The buffer should be large enough to hold binary data of the size *N* pages (specified in the DB_BACKUP_START_PARALLEL operation, or in a DB_BACKUP_INFO operation). For more information about this data type, see DT_LONGBINARY in "Embedded SQL data types" on page 518.

  The server returns a sequential block of database pages for a particular database file. The page number of the first page in the block is returned in the SQLCOUNT field. The file number that the pages belong to is returned in the SQLIOESTIMATE field, and this value matches one of the file numbers used in the DB_BACKUP_OPEN_FILE calls. The size of the data returned is available in the *stored_len* field of the DT_LONGBINARY variable, and is always a multiple of the database page size. While the data returned by this call contains a block of sequential pages for a given file, it is not safe to assume that separate blocks of data are returned in sequential order, or that all of one database file's pages are returned before another database file's pages. The caller should be prepared to receive portions of another individual file out of sequential order, or of any opened database file on any given call.

  An application should make repeated calls to this operation until the size of the read data is 0, or the value of sqlda->sqlvar[0].sqlind is greater than 0. If the backup is started with transaction log renaming/restarting, SQLERROR could be set to SQLE_BACKUP_CANNOT_RENAME_LOG_YET. In this case, do not use the pages returned, but instead reissue the request until you receive SQLE_NOERROR, and then write the data. The SQLE_BACKUP_CANNOT_RENAME_LOG_YET error may be returned multiple times and on multiple pages. In your retry loop, you should add a delay so the database server is not slowed down by too many requests. Continue reading the pages until either of the first two conditions are met.

The dbbackup utility uses the following algorithm. Note that this is *not* C code, and does not include error checking.


```
sqlda->sqld = 1;
sqlda->sqlvar[0].sqltype = DT_LONGBINARY

/* Allocate LONGBINARY value for page buffer. It MUST have */
/* enough room to hold the requested number (128) of database pages */
sqlda->sqlvar[0].sqldata = allocated buffer

/* Open the server files needing backup */
for file_num = 0 to DB_BACKUP_MAX_FILE
  db_backup( ... DB_BACKUP_OPEN_FILE, file_num ... )
  if SQLCODE == SQLE_NO_ERROR
    /* The file exists */
    num_pages = SQLCOUNT
    file_time = SQLE_IO_ESTIMATE
    open backup file with name from sqlca.sqlerrmc
end for

/* read pages from the server, write them locally */
while TRUE
  /* file_no and page_no are ignored */
  db_backup( &sqlca, DB_BACKUP_PARALLEL_READ, 0, 0, &sqlda );

  if SQLCODE != SQLE_NO_ERROR
     break;

  if buffer->stored_len == 0 || sqlda->sqlvar[0].sqlind > 0
     break;

  /* SQLCOUNT contains the starting page number of the block    */
  /* SQLIOESTIMATE contains the file number the pages belong to */
  write block of pages to appropriate backup file
end while

/* close the server backup files */
for file_num = 0 to DB_BACKUP_MAX_FILE
    /* close backup file */
    db_backup( ... DB_BACKUP_CLOSE_FILE, file_num ... )
end for

/* shut down the backup */
db_backup( ... DB_BACKUP_END ... )

/* cleanup */
free page buffer
```

db_cancel_request function

Prototype

int **db_cancel_request(** SQLCA * *sqlca* **);**

Description

Cancels the currently active database server request. This function checks to make sure a database server request is active before sending the cancel request. If the function returns 1, then the cancel request was sent; if it returns 0, then no request was sent.

A non-zero return value does not mean that the request was canceled. There are a few critical timing cases where the cancel request and the response from the database or server cross. In these cases, the cancel simply has no effect, even though the function still returns TRUE.

The db_cancel_request function can be called asynchronously. This function and db_is_working are the only functions in the database interface library that can be called asynchronously using a SQLCA that might be in use by another request.

If you cancel a request that is carrying out a cursor operation, the position of the cursor is indeterminate. You must locate the cursor by its absolute position or close it, following the cancel.

db_change_char_charset function

Prototype

unsigned int **db_change_char_charset(**
SQLCA * *sqlca*,
char * charset **);**

Description

Changes the application's CHAR character set for this connection. Data sent and fetched using FIXCHAR, VARCHAR, LONGVARCHAR, and STRING types are in the CHAR character set.

Returns 1 if the change is successful, 0 otherwise.

For a list of recommended character sets, see "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*].

db_change_nchar_charset function

Prototype

unsigned int **db_change_nchar_charset(**
SQLCA * *sqlca*,
char * charset **);**

Description

Changes the application's NCHAR character set for this connection. Data sent and fetched using NFIXCHAR, NVARCHAR, LONGNVARCHAR, and NSTRING host variable types are in the NCHAR character set.

If the db_change_nchar_charset function is not called, all data is sent and fetched using the CHAR character set. Typically, an application that wants to send and fetch Unicode data should set the NCHAR character set to UTF-8.

If this function is called, the charset parameter is usually "UTF-8". The NCHAR character set cannot be set to UTF-16.

Returns 1 if the change is successful, 0 otherwise.

In embedded SQL, NCHAR, NVARCHAR and LONG NVARCHAR are described as DT_FIXCHAR, DT_VARCHAR, and DT_LONGVARCHAR, respectively, by default. If the db_change_nchar_charset function has been called, these types are described as DT_NFIXCHAR, DT_NVARCHAR, and DT_LONGNVARCHAR, respectively.

For a list of recommended character sets, see "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*].

db_delete_file function

Prototype

void **db_delete_file(**
SQLCA * *sqlca*,
char * *filename* **);**

Authorization

Must be connected to a user ID with DBA authority or REMOTE DBA authority (SQL Remote).

Description

The db_delete_file function requests the database server to delete *filename*. This can be used after backing up and renaming the transaction log to delete the old transaction log. See DB_BACKUP_READ_RENAME_LOG in "db_backup function" on page 569.

You must be connected to a user ID with DBA authority.

db_find_engine function

Prototype

unsigned short **db_find_engine(**
SQLCA * *sqlca*,
char * *name* **);**

Description

Returns an unsigned short value, which indicates status information about the local database server whose name is *name*. If no server can be found over shared memory with the specified name, the return value is 0. A non-zero value indicates that the local server is currently running.

If a null pointer is specified for *name*, information is returned about the default database server.

Each bit in the return value conveys some information. Constants that represent the bits for the various pieces of information are defined in the *sqldef.h* header file. Their meaning is described below.

- **DB_ENGINE** This flag is always set.
- **DB_CLIENT** This flag is always set.
- **DB_CAN_MULTI_DB_NAME** This flag is obsolete.

- **DB_DATABASE_SPECIFIED** This flag is always set.
- **DB_ACTIVE_CONNECTION** This flag is always set.
- **DB_CONNECTION_DIRTY** This flag is obsolete.
- **DB_CAN_MULTI_CONNECT** This flag is obsolete.
- **DB_NO_DATABASES** This flag is set if the server has no databases started.

db_fini function

Prototype

int **db_fini(** SQLCA * *sqlca* **);**

Description

This function frees resources used by the database interface or DLL. You must not make any other library calls or execute any embedded SQL statements after db_fini is called. If an error occurs during processing, the error code is set in SQLCA and the function returns 0. If there are no errors, a non-zero value is returned.

You need to call db_fini once for each SQLCA being used.

For information about using db_init in UltraLite applications, see "db_fini function" [*UltraLite - C and C++ Programming*].

db_get_property function

Prototype

unsigned int **db_get_property(**
SQLCA * *sqlca***,**
a_db_property *property***,**
char * *value_buffer***,**
int *value_buffer_size* **);**

Description

This function is used to obtain information about the database interface or the server to which you are connected.

The arguments are as follows:

- **a_db_property** The property requested, either DB_PROP_CLIENT_CHARSET, DB_PROP_SERVER_ADDRESS, or DB_PROP_DBLIB_VERSION.
- **value_buffer** This argument is filled with the property value as a null-terminated string.
- **value_buffer_size** The maximum length of the string value_buffer, including the terminating null character.

The following properties are supported:

- **DB_PROP_CLIENT_CHARSET** This property value gets the client character set (for example, "windows-1252").
- **DB_PROP_SERVER_ADDRESS** This property value gets the current connection's server network address as a printable string. The shared memory protocol always returns the empty string for the address. The TCP/IP protocol returns non-empty string addresses.
- **DB_PROP_DBLIB_VERSION** This property value gets the database interface library's version (for example, "11.0.0.1297").

Returns 1 if successful, 0 otherwise.

db_init function

Prototype

int **db_init(** SQLCA * *sqlca* **);**

Description

This function initializes the database interface library. This function must be called before any other library call is made and before any embedded SQL statement is executed. The resources the interface library required for your program are allocated and initialized on this call.

Returns 1 if successful, 0 otherwise.

Use db_fini to free the resources at the end of your program. If there are any errors during processing, they are returned in the SQLCA and 0 is returned. If there are no errors, a non-zero value is returned and you can begin using embedded SQL statements and functions.

In most cases, this function should be called only once (passing the address of the global sqlca variable defined in the *sqlca.h* header file). If you are writing a DLL or an application that has multiple threads using embedded SQL, call db_init once for each SQLCA that is being used.

For more information, see "SQLCA management for multi-threaded or reentrant code" on page 533.

For information about using db_init in UltraLite applications, see "db_init function" [*UltraLite - C and C++ Programming*].

db_is_working function

Prototype

unsigned short **db_is_working(** SQLCA * *sqlca* **);**

Description

Returns 1 if your application has a database request in progress that uses the given sqlca and 0 if there is no request in progress that uses the given sqlca.

This function can be called asynchronously. This function and db_cancel_request are the only functions in the database interface library that can be called asynchronously using a SQLCA that might be in use by another request.

db_locate_servers function

Prototype

unsigned int**db_locate_servers(**
SQLCA * *sqlca***,**
SQL_CALLBACK_PARM *callback_address***,**
void * *callback_user_data* **);**

Description

Provides programmatic access to the information displayed by the dblocate utility, listing all the SQL Anywhere database servers on the local network that are listening on TCP/IP.

The callback function must have the following prototype:

int (*)(SQLCA * *sqlca*,
a_server_address * *server_addr*,
void * *callback_user_data*);

The callback function is called for each server found. If the callback function returns 0, db_locate_servers stops iterating through servers.

The sqlca and callback_user_data passed to the callback function are those passed into db_locate_servers. The second parameter is a pointer to an a_server_address structure. a_server_address is defined in *sqlca.h*, with the following definition:

```
typedef struct a_server_address {
    a_sql_uint32 port_type;
    a_sql_uint32 port_num;
    char         *name;
    char         *address;
} a_server_address;
```

- **port_type** Is always PORT_TYPE_TCP at this time (defined to be 6 in *sqlca.h*).
- **port_num** Is the TCP port number on which this server is listening.
- **name** Points to a buffer containing the server name.
- **address** Points to a buffer containing the IP address of the server.

Returns 1 if successful, 0 otherwise.

See also

- "Server Enumeration utility (dblocate)" [*SQL Anywhere Server - Database Administration*]

db_locate_servers_ex function

Prototype

unsigned int **db_locate_servers_ex(**
SQLCA * *sqlca*,
SQL_CALLBACK_PARM *callback_address*,
void * *callback_user_data*,
unsigned int *bitmask*);

Description

Provides programmatic access to the information displayed by the dblocate utility, listing all the SQL Anywhere database servers on the local network that are listening on TCP/IP, and provides a mask parameter used to select addresses passed to the callback function.

The callback function must have the following prototype:

int (*)(SQLCA * *sqlca*,
a_server_address * *server_addr*,
void * *callback_user_data*);

The callback function is called for each server found. If the callback function returns 0, db_locate_servers_ex stops iterating through servers.

The sqlca and callback_user_data passed to the callback function are those passed into db_locate_servers. The second parameter is a pointer to an a_server_address structure. a_server_address is defined in *sqlca.h*, with the following definition:

```
typedef struct a_server_address {
    a_sql_uint32   port_type;
    a_sql_uint32   port_num;
    char           *name;
    char           *address;
    char           *dbname;
} a_server_address;
```

- **port_type** Is always PORT_TYPE_TCP at this time (defined to be 6 in *sqlca.h*).
- **port_num** Is the TCP port number on which this server is listening.
- **name** Points to a buffer containing the server name.
- **address** Points to a buffer containing the IP address of the server.
- **dbname** Points to a buffer containing the database name.

Three bitmask flags are supported:

- DB_LOOKUP_FLAG_NUMERIC
- DB_LOOKUP_FLAG_ADDRESS_INCLUDES_PORT
- DB_LOOKUP_FLAG_DATABASES

These flags are defined in *sqlca.h* and can be ORed together.

DB_LOOKUP_FLAG_NUMERIC ensures that addresses passed to the callback function are IP addresses, instead of host names.

DB_LOOKUP_FLAG_ADDRESS_INCLUDES_PORT specifies that the address includes the TCP/IP port number in the a_server_address structure passed to the callback function.

DB_LOOKUP_FLAG_DATABASES specifies that the callback function is called once for each database found, or once for each database server found if the database server doesn't support sending database information (version 9.0.2 and earlier database servers).

Returns 1 if successful, 0 otherwise.

For more information, see "Server Enumeration utility (dblocate)" [*SQL Anywhere Server - Database Administration*].

db_register_a_callback function

Prototype

void **db_register_a_callback(**
SQLCA * *sqlca***,**
a_db_callback_index *index***,**
(SQL_CALLBACK_PARM) *callback* **);**

Description

This function registers callback functions.

If you do not register a DB_CALLBACK_WAIT callback, the default action is to do nothing. Your application blocks, waiting for the database response. You must register a callback for the MESSAGE TO CLIENT statement. See "MESSAGE statement" [*SQL Anywhere Server - SQL Reference*].

To remove a callback, pass a null pointer as the *callback* function.

The following values are allowed for the *index* parameter:

- **DB_CALLBACK_DEBUG_MESSAGE** The supplied function is called once for each debug message and is passed a null-terminated string containing the text of the debug message. A debug message is a message that is logged to the LogFile file. In order for a debug message to be passed to this callback, the LogFile connection parameter must be used. The string normally has a newline character (\n) immediately before the terminating null character. The prototype of the callback function is as follows:

  void SQL_CALLBACK *debug_message_callback***(**
  SQLCA * *sqlca*,
  char * *message_string* **);**

  For more information, see "LogFile connection parameter [LOG]" [*SQL Anywhere Server - Database Administration*].

- **DB_CALLBACK_START** The prototype is as follows:

  void SQL_CALLBACK *start_callback***(** SQLCA * *sqlca* **);**

  This function is called just before a database request is sent to the server. DB_CALLBACK_START is used only on Windows.

- **DB_CALLBACK_FINISH** The prototype is as follows:

void SQL_CALLBACK *finish_callback***(** SQLCA * *sqlca* **);**

This function is called after the response to a database request has been received by the interface DLL. DB_CALLBACK_FINISH is used only on Windows operating systems.

- **DB_CALLBACK_CONN_DROPPED** The prototype is as follows:

  void SQL_CALLBACK *conn_dropped_callback* **(**
  SQLCA * *sqlca*,
  char * *conn_name* **);**

  This function is called when the database server is about to drop a connection because of a liveness timeout, through a DROP CONNECTION statement, or because the database server is being shut down. The connection name *conn_name* is passed in to allow you to distinguish between connections. If the connection was not named, it has a value of NULL.

- **DB_CALLBACK_WAIT** The prototype is as follows:

  void SQL_CALLBACK *wait_callback***(** SQLCA * *sqlca* **);**

  This function is called repeatedly by the interface library while the database server or client library is busy processing your database request.

  You would register this callback as follows:

  ```
  db_register_a_callback( &sqlca,
     DB_CALLBACK_WAIT,
     (SQL_CALLBACK_PARM)&db_wait_request );
  ```

- **DB_CALLBACK_MESSAGE** This is used to enable the application to handle messages received from the server during the processing of a request.

  The callback prototype is as follows:

  void SQL_CALLBACK *message_callback***(**
  SQLCA * *sqlca*,
  unsigned char *msg_type*,
  an_sql_code *code*,
  unsigned short *length*,
  char * *msg*
  );

  The *msg_type* parameter states how important the message is and you may want to handle different message types in different ways. The available message types are MESSAGE_TYPE_INFO, MESSAGE_TYPE_WARNING, MESSAGE_TYPE_ACTION, and MESSAGE_TYPE_STATUS. These constants are defined in *sqldef.h*. The *code* field may provide a SQLCODE associated with the message, otherwise the value is 0. The *length* field tells you how long the message is. The message is *not* null-terminated.

  For example, the Interactive SQL callback displays STATUS and INFO message on the Messages tab, while messages of type ACTION and WARNING go to a window. If an application does not register this callback, there is a default callback, which causes all messages to be written to the server logfile (if debugging is on and a logfile is specified). In addition, messages of type MESSAGE_TYPE_WARNING and MESSAGE_TYPE_ACTION are more prominently displayed, in an operating system-dependent manner.

- **DB_CALLBACK_VALIDATE_FILE_TRANSFER** This is used to register a file transfer validation callback function. Before allowing any transfer to take place, the client library will invoke the validation callback, if it exists. If the client data transfer is being requested during the execution of indirect statements such as from within a stored procedure, the client library will not allow a transfer unless the client application has registered a validation callback. The conditions under which a validation call is made are described more fully below.

  The callback prototype is as follows:

  int SQL_CALLBACK *file_transfer_callback*(
  SQLCA * *sqlca*,
  char * *file_name*,
  int *is_write*
  );

  The *file_name* parameter is the name of the file to be read or written. The *is_write* parameter is 0 if a read is requested (transfer from the client to the server), and non-zero for a write. The callback function should return 0 if the file transfer is not allowed, non-zero otherwise.

  For data security, the server tracks the origin of statements requesting a file transfer. The server determines if the statement was received directly from the client application. When initiating the transfer of data from the client, the server sends the information about the origin of the statement to the client software. On its part, the embedded SQL client library allows unconditional transfer of data only if the data transfer is being requested due to the execution of a statement sent directly by the client application. Otherwise, the application must have registered the validation callback described above, in the absence of which the transfer is denied and the statement fails with an error. Note that if the client statement invokes a stored procedure already existing in the database, then the execution of the stored procedure itself is considered not to have been for a client initiated statement. However, if the client application explicitly creates a temporary stored procedure then the execution of the stored procedure results in the server treating the procedure as having been client initiated. Similarly, if the client application executes a batch statement, then the execution of the batch statement is considered as being done directly by the client application.

db_start_database function

Prototype

unsigned int **db_start_database(** SQLCA * *sqlca*, char * *parms* **);**

Arguments

- **sqlca** A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 531.
- **parms** A null-terminated string containing a semicolon-delimited list of parameter settings, each of the form KEYWORD=*value*. For example:

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Description

The database is started on an existing server, if possible. Otherwise, a new server is started. The steps carried out to start a database are described in "Locating a database server" [*SQL Anywhere Server - Database Administration*].

If the database was already running or was successfully started, the return value is true (non-zero) and SQLCODE is set to 0. Error information is returned in the SQLCA.

If a user ID and password are supplied in the parameters, they are ignored.

The permission required to start and stop a database is set on the server command line. For information, see "-gd server option" [*SQL Anywhere Server - Database Administration*].

db_start_engine function

Prototype

unsigned int **db_start_engine(** SQLCA * *sqlca*, char * *parms* **);**

Arguments

- **sqlca** A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 531.
- **parms** A null-terminated string containing a semicolon-delimited list of parameter settings, each of the form KEYWORD=*value*. For example,

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

  For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Description

Starts the database server if it is not running.

For a description of the steps carried out by this function, see "Locating a database server" [*SQL Anywhere Server - Database Administration*].

If the database server was already running or was successfully started, the return value is TRUE (non-zero) and SQLCODE is set to 0. Error information is returned in the SQLCA.

The following call to db_start_engine starts the database server and names it demo, but does not load the database, despite the DBF connection parameter:

```
db_start_engine( &sqlca,
    "DBF=samples-dir\\demo.db;START=dbeng11" );
```

If you want to start a database as well as the server, include the database file in the StartLine (START) connection parameter:

```
db_start_engine( &sqlca,
   "ENG=eng_name;START=dbeng11 samples-dir\\demo.db" );
```

This call starts the server, names it eng_name, and starts the SQL Anywhere sample database on that server.

The db_start_engine function attempts to connect to a server before starting one, to avoid attempting to start a server that is already running.

The ForceStart (FORCE) connection parameter is used only by the db_start_engine function. When set to YES, there is no attempt to connect to a server before trying to start one. This enables the following pair of commands to work as expected:

1. Start a database server named server_1:

```
start dbeng11 -n server_1 demo.db
```

2. Force a new server to start and connect to it:

```
db_start_engine( &sqlda,
  "START=dbeng11 -n server_2 mydb.db;ForceStart=YES" )
```

If ForceStart (FORCE) was not used, and without a ServerName (ENG) parameter, the second command would have attempted to connect to server_1. The db_start_engine function does not pick up the server name from the -n option of the StartLine (START) parameter.

db_stop_database function

Prototype

unsigned int **db_stop_database(** SQLCA * *sqlca***,** char * *parms* **);**

Arguments

- **sqlca** A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 531.
- **parms** A null-terminated string containing a semicolon-delimited list of parameter settings, each of the form KEYWORD=*value*. For example:

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

  For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Description

Stop the database identified by DatabaseName (DBN) on the server identified by ServerName (ENG). If ServerName is not specified, the default server is used.

By default, this function does not stop a database that has existing connections. If Unconditional is yes, the database is stopped regardless of existing connections.

A return value of TRUE indicates that there were no errors.

The permission required to start and stop a database is set on the server command line. For information, see "-gd server option" [*SQL Anywhere Server - Database Administration*].

db_stop_engine function

Prototype

unsigned int **db_stop_engine(** SQLCA * *sqlca*, char * *parms* **);**

Arguments

- **sqlca** A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 531.
- **parms** A null-terminated string containing a semicolon-delimited list of parameter settings, each of the form KEYWORD=*value*. For example,

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

  For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Description

Stops execution of the database server. The steps carried out by this function are:

- Look for a local database server that has a name that matches the ServerName (ENG) parameter. If no ServerName is specified, look for the default local database server.
- If no matching server is found, this function returns with success.
- Send a request to the server to tell it to checkpoint and shut down all databases.
- Unload the database server.

By default, this function does not stop a database server that has existing connections. If Unconditional is yes, the database server is stopped regardless of existing connections.

A C program can use this function instead of spawning dbstop. A return value of TRUE indicates that there were no errors.

The use of db_stop_engine is subject to the permissions set with the -gk server option. See "-gk server option" [*SQL Anywhere Server - Database Administration*].

db_string_connect function

Prototype

unsigned int **db_string_connect(** SQLCA * *sqlca*, char * *parms* **);**

Arguments

- **sqlca** A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 531.
- **parms** A null-terminated string containing a semicolon-delimited list of parameter settings, each of the form KEYWORD=*value*. For example:

```
"UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
```

For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Description

Provides extra functionality beyond the embedded SQL CONNECT statement.

For a description of the algorithm used by this function, see "Troubleshooting connections" [*SQL Anywhere Server - Database Administration*].

The return value is TRUE (non-zero) if a connection was successfully established and FALSE (zero) otherwise. Error information for starting the server, starting the database, or connecting is returned in the SQLCA.

db_string_disconnect function

Prototype

unsigned int **db_string_disconnect(**
SQLCA * *sqlca*,
char * *parms* **);**

Arguments

- **sqlca** A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 531.
- **parms** A null-terminated string containing a semicolon-delimited list of parameter settings, each of the form KEYWORD=*value*. For example:

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

  For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Description

This function disconnects the connection identified by the ConnectionName parameter. All other parameters are ignored.

If no ConnectionName parameter is specified in the string, the unnamed connection is disconnected. This is equivalent to the embedded SQL DISCONNECT statement. The return value is TRUE if a connection was successfully ended. Error information is returned in the SQLCA.

This function shuts down the database if it was started with the AutoStop=yes parameter and there are no other connections to the database. It also stops the server if it was started with the AutoStop=yes parameter and there are no other databases running.

db_string_ping_server function

Prototype

unsigned int **db_string_ping_server(**
SQLCA * *sqlca*,
char * *connect_string*,
unsigned int *connect_to_db* **);**

Description

- **connect_string** The *connect_string* is a normal connect string that may or may not contain server and database information.
- **connect_to_db** If *connect_to_db* is non-zero (TRUE), then the function attempts to connect to a database on a server. It returns TRUE only if the connect string is sufficient to connect to the named database on the named server.
- **connect_to_db** If *connect_to_db* is zero, then the function only attempts to locate a server. It returns TRUE only if the connect string is sufficient to locate a server. It makes no attempt to connect to the database.

db_time_change function

Prototype

unsigned int **db_time_change(**
SQLCA * *sqlca*);

Description

sqlca A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 531.
This function permits clients to notify the server that the time has changed on the client. This function recalculates the time zone adjustment and sends it to the server. On Windows platforms, it is recommended that applications call this function when they receive the WM_TIMECHANGE message. This will make sure that UTC timestamps are consistent over time changes, time zone changes, or daylight savings time changeovers.

Returns TRUE if successful, and FALSE otherwise.

fill_s_sqlda function

Prototype

struct sqlda * **fill_s_sqlda(**
struct sqlda * *sqlda*,
unsigned int *maxlen* **);**

Description

The same as fill_sqlda, except that it changes all the data types in *sqlda* to type DT_STRING. Enough space is allocated to hold the string representation of the type originally specified by the SQLDA, up to a maximum of *maxlen* bytes. The length fields in the SQLDA (sqllen) are modified appropriately. Returns *sqlda* if successful and returns the null pointer if there is not enough memory available.

The SQLDA should be freed using the free_filled_sqlda function.

fill_sqlda function

Prototype

struct sqlda * **fill_sqlda(** struct sqlda * *sqlda* **);**

Description

Allocates space for each variable described in each descriptor of *sqlda*, and assigns the address of this memory to the sqldata field of the corresponding descriptor. Enough space is allocated for the database type and length indicated in the descriptor. Returns *sqlda* if successful and returns the null pointer if there is not enough memory available.

The SQLDA should be freed using the free_filled_sqlda function.

free_filled_sqlda function

Prototype

void **free_filled_sqlda(** struct sqlda * *sqlda* **);**

Description

Free the memory allocated to each sqldata pointer and the space allocated for the SQLDA itself. Any null pointer is not freed.

This should only be called if fill_sqlda or fill_s_sqlda was used to allocate the sqldata fields of the SQLDA.

Calling this function causes free_sqlda to be called automatically, and so any descriptors allocated by alloc_sqlda are freed.

free_sqlda function

Prototype

void **free_sqlda(** struct sqlda * *sqlda* **);**

Description

Free space allocated to this *sqlda* and free the indicator variable space, as allocated in fill_sqlda. Do not free the memory referenced by each sqldata pointer.

free_sqlda_noind function

Prototype

void **free_sqlda_noind(** struct sqlda * *sqlda* **);**

Description

Free space allocated to this *sqlda*. Do not free the memory referenced by each sqldata pointer. The indicator variable pointers are ignored.

sql_needs_quotes function

Prototype

unsigned int **sql_needs_quotes(** SQLCA **sqlca*, char * *str* **);**

Description

Returns a TRUE or FALSE value that indicates whether the string requires double quotes around it when it is used as a SQL identifier. This function formulates a request to the database server to determine if quotes are needed. Relevant information is stored in the sqlcode field.

There are three cases of return value/code combinations:

- **return = FALSE, sqlcode = 0** The string does not need quotes.
- **return = TRUE** The sqlcode is always SQLE_WARNING, and the string requires quotes.
- **return = FALSE** If sqlcode is something other than SQLE_WARNING, the test is inconclusive.

sqlda_storage function

Prototype

unsigned int **sqlda_storage(** struct sqlda * *sqlda*, int *varno* **);**

Description

Returns an unsigned 32-bit integer value representing the amount of storage required to store any value for the variable described in sqlda->sqlvar[varno].

sqlda_string_length function

Prototype

unsigned int **sqlda_string_length(** struct sqlda * *sqlda*, int *varno* **);**

Description

Returns an unsigned 32-bit integer value representing the length of the C string (type DT_STRING) that would be required to hold the variable sqlda->sqlvar[varno] (no matter what its type is).

sqlerror_message function

Prototype

char * **sqlerror_message(** SQLCA * *sqlca***,** char * *buffer***,** int *max* **);**

Description

Return a pointer to a string that contains an error message. The error message contains text for the error code in the SQLCA. If no error was indicated, a null pointer is returned. The error message is placed in the buffer supplied, truncated to length *max* if necessary.

Embedded SQL statement summary

EXEC SQL
ALL embedded SQL statements must be preceded with EXEC SQL and end with a semicolon (;).

There are two groups of embedded SQL statements. Standard SQL statements are used by simply placing them in a C program enclosed with EXEC SQL and a semicolon (;). CONNECT, DELETE, SELECT, SET, and UPDATE have additional formats only available in embedded SQL. The additional formats fall into the second category of embedded SQL specific statements.

For descriptions of the standard SQL statements, see "SQL statements" [*SQL Anywhere Server - SQL Reference*].

Several SQL statements are specific to embedded SQL and can only be used in a C program. See "SQL language elements" [*SQL Anywhere Server - SQL Reference*].

Standard data manipulation and data definition statements can be used from embedded SQL applications. In addition the following statements are specifically for embedded SQL programming:

- **ALLOCATE DESCRIPTOR** allocate memory for a descriptor. See "ALLOCATE DESCRIPTOR statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **CLOSE** close a cursor. See "CLOSE statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].
- **CONNECT** connect to the database. See "CONNECT statement [ESQL] [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].
- **DEALLOCATE DESCRIPTOR** reclaim memory for a descriptor. See "DEALLOCATE DESCRIPTOR statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **Declaration section** declare host variables for database communication. See "Declaration section [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **DECLARE CURSOR** declare a cursor. See "DECLARE CURSOR statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].
- **DELETE (positioned)** delete the row at the current position in a cursor. See "DELETE (positioned) statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].
- **DESCRIBE** describe the host variables for a particular SQL statement. See "DESCRIBE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **DISCONNECT** disconnect from database server. See "DISCONNECT statement [ESQL] [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].
- **DROP STATEMENT** free resources used by a prepared statement. See "DROP STATEMENT statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **EXECUTE** execute a particular SQL statement. See "EXECUTE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **EXPLAIN** explain the optimization strategy for a particular cursor. See "EXPLAIN statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

- **FETCH** fetch a row from a cursor. See "FETCH statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].
- **GET DATA** fetch long values from a cursor. See "GET DATA statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **GET DESCRIPTOR** retrieve information about a variable in a SQLDA. See "GET DESCRIPTOR statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **GET OPTION** get the setting for a particular database option. See "GET OPTION statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **INCLUDE** include a file for SQL preprocessing. See "INCLUDE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **OPEN** open a cursor. See "OPEN statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].
- **PREPARE** prepare a particular SQL statement. See "PREPARE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **PUT** insert a row into a cursor. See "PUT statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **SET CONNECTION** change active connection. See "SET CONNECTION statement [Interactive SQL] [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **SET DESCRIPTOR** describe the variables in a SQLDA and place data into the SQLDA. See "SET DESCRIPTOR statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **SET SQLCA** use a SQLCA other than the default global one. See "SET SQLCA statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].
- **UPDATE (positioned)** update the row at the current location of a cursor. See "UPDATE (positioned) statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].
- **WHENEVER** specify actions to occur on errors in SQL statements. See "WHENEVER statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

CHAPTER 15

SQL Anywhere External Function API

Contents

Calling external libraries from procedures

You can call a function in an external library from a stored procedure or function. You can call functions in a DLL under Windows operating systems and in a shared object on Unix. You cannot call external functions on Windows Mobile.

This section describes how to use the external library call API. Sample external stored procedures, plus the files required to build a DLL containing them, are located in the following folder: *samples-dir\SQLAnywhere \ExternalProcedures*. For information about the location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

Caution

External libraries called from procedures share the memory of the server. If you call an external library from a procedure and the external library contains memory-handling errors, you can crash the server or corrupt your database. Ensure that you thoroughly test your libraries before deploying them on production databases.

The API described in this section replaces an older API. The older API is deprecated. Libraries written to the older API, used in versions before version 7.0.x, are still supported, but in any new development, we encourage you to use the new API. Note that the new API must be used for all Unix platforms and for all 64-bit platforms, including 64-bit Windows.

SQL Anywhere includes a set of system procedures that make use of this capability, for example to send MAPI email messages. See "MAPI and SMTP procedures" [*SQL Anywhere Server - SQL Reference*].

Creating procedures and functions with external calls

This section presents some examples of procedures and functions with external calls.

DBA authority required
You must have DBA authority to create procedures or functions that reference external libraries. This requirement is more strict than the RESOURCE authority required for creating other procedures or functions.

Syntax

You can create a procedure that calls a function *function-name* in the Windows Dynamic Link Library *library.dll* as follows:

```
CREATE PROCEDURE dll_proc( parameter-list )
  EXTERNAL NAME 'function-name@library.dll';
```

When you define a stored procedure or function in this way, you are creating a bridge to the function in the external DLL. The stored procedure or function cannot perform any other tasks.

An analogous CREATE FUNCTION statement is as follows:

```
CREATE FUNCTION dll_func ( parameter-list )
  RETURNS data-type
  EXTERNAL NAME 'function-name@library.dll';
```

In these statements, *function-name* is the exported name of a function in the DLL, and *library.dll* is the name of the library. The arguments in *parameter-list* must correspond in type and order to the arguments expected by the library function. The library function accesses the procedure arguments using an API described in "External function prototypes" on page 597.

Any value returned by the external function can be returned by the stored function or procedure to the calling environment.

No other statements permitted

A stored procedure or function that references an external function can include no other statements: its sole purpose is to take arguments for a function, call the function, and return any value and returned arguments from the function to the calling environment. You can use IN, INOUT, or OUT parameters in the procedure call in the same way as for other procedures: the input values get passed to the external function, and any parameters modified by the function are returned to the calling environment in OUT or INOUT parameters or as the RETURNS result of the stored function.

System-dependent calls

You can specify operating-system dependent calls, so that a procedure calls one function when run on one operating system, and another function (presumably analogous) on another operating system. The syntax for such calls involves prefixing the function name with the operating system name. The operating system identifier must be Unix. An example follows.

```
CREATE FUNCTION func ( parameter-list )
  RETURNS data-type
  EXTERNAL NAME 'Unix:function-name@library.so;function-name@library.dll';
```

If the list of functions does not contain an entry for the operating system on which the server is running, but the list does contain an entry without an operating system specified, the database server calls the function in that entry.

See also

- "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE FUNCTION statement" [*SQL Anywhere Server - SQL Reference*]

External function prototypes

This section describes the API for functions in external libraries.

The API is defined by a header file named *extfnapi.h*, in the *SDK\Include* subdirectory of your SQL Anywhere installation directory. This header file handles the platform-dependent features of external function prototypes. The API supercedes a previous API for functions in external libraries.

Declaring the API version

To notify the database server that the external library is written using the new API, your external library must export the following function as follows:

a_sql_uint32 extfn_use_new_api( )

The function returns an unsigned 32-bit integer. The returned value must be the API version number, EXTFN_API_VERSION, defined in *extfnapi.h*. A return value of 0 means that the old API is being used.

If the function is not exported by the library, the database server assumes that the old API is in use. The new API must be used for all Unix platforms and for all 64-bit platforms, including 64-bit Windows.

A typical implementation of this function follows:

```
a_sql_uint32 extfn_use_new_api( void )
{
    return( EXTFN_API_VERSION );
}
```

Declaring the optional cancel processing routine

To notify the database server that the external library supports cancel processing, your external library must export the following function as follows:

void extfn_cancel(void *cancel_handle)

The function uses the cancel_handle to set a flag indicating to the external library functions that the SQL statement has been canceled.

If the function is not exported by the library, the database server assumes that cancel processing is not supported.

A typical implementation of this function follows:

```
void extfn_cancel( void *cancel_handle )
{
    *(short *)cancel_handle = 1;
}
```

Function prototypes

The name of the function must match that referenced in the CREATE PROCEDURE or CREATE FUNCTION statement. The function declaration must be as follows:

void *function-name***(an_extfn_api ***api**, void ***argument-handle **)**

The function must return void, and must take as arguments a pointer to a structure used to call a set of callback functions and a handle to the arguments provided by the SQL procedure.

The an_extfn_api structure has the following form:

```
typedef struct an_extfn_api {
    short (SQL_CALLBACK *get_value)(
                void *          arg_handle,
                a_sql_uint32    arg_num,
                an_extfn_value *value
                );
    short (SQL_CALLBACK *get_piece)(
                void *          arg_handle,
                a_sql_uint32    arg_num,
                an_extfn_value *value,
                a_sql_uint32    offset
                );
    short (SQL_CALLBACK *set_value)(
                void *          arg_handle,
                a_sql_uint32    arg_num,
                an_extfn_value *value
                short           append
                );
    void (SQL_CALLBACK *set_cancel)(
                void *          arg_handle,
                void *          cancel_handle
                );
} an_extfn_api;
```

The an_extfn_value structure has the following form:

```
typedef struct an_extfn_value {
    void *           data;
    a_sql_uint32     piece_len;
    union {
        a_sql_uint32 total_len;
        a_sql_uint32 remain_len;
    } len;
    a_sql_data_type  type;
} an_extfn_value;
```


Using the external function call API routines

```
short (SQL_CALLBACK *get_value)
(
   void *            arg_handle,
   a_sql_uint32      arg_num,
   an_extfn_value *value
);
```

The get_value callback function can be used to obtain the value of a parameter that was passed to the stored procedure or function that acts as the interface to the external function. It returns 0 if not successful; otherwise it returns a non-zero result. After calling get_value, the total_len field of the an_extfn_value structure contains the length of the entire value. The piece_len field contains the length of the portion that was obtained as a result of calling get_value. Note that piece_len will always be less than or equal to total_len. When it is less than, a second function get_piece can be called to obtain the remaining pieces. Note that the total_len field is only valid after the initial call to get_value. This field is overlaid by the remain_len field which is altered by calls to get_piece. It is important to preserve the value of the total_len field immediately after calling get_value if you plan to use it later on.

```
short (SQL_CALLBACK *get_piece)
(
   void *            arg_handle,
   a_sql_uint32      arg_num,
   an_extfn_value *value,
   a_sql_uint32      offset
);
```

If the entire parameter value cannot be returned in one piece, then a second function, get_piece, can be called iteratively to obtain the remaining pieces of the parameter value.

The sum of all the piece_len values returned by both calls to get_value and get_piece will add up to the initial value that was returned in the total_len field after calling get_value. After calling get_piece, the remain_len field, which overlays total_len, represents the amount not yet obtained.

The following example shows the use of get_value and get_piece to obtain the value of a string parameter such as a long varchar parameter.

Suppose that the wrapper to an external function was declared as follows:

```
CREATE PROCEDURE mystring( IN instr LONG VARCHAR )
  EXTERNAL NAME 'mystring@mystring.dll';
```

To call the external function from SQL, we would use a statement like the following.

```
call mystring('Hello world!');
```

A sample implementation for the Windows operating system of the mystring function, written in C, follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
```

```
                        DWORD  ul_reason_for_call,
                        LPVOID lpReserved
      )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
a_sql_uint32 extfn_use_new_api( void )
{
    return( EXTFN_API_VERSION );
}

extern "C" __declspec( dllexport )
void mystring( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    unsigned            offset;
    char                *string;

    result = api->get_value( arg_handle, 1, &arg );
    if( result == 0 || arg.data == NULL )
    {
        return; // no parameter or parameter is NULL
    }
    string = (char *)malloc( arg.len.total_len + 1 );
    offset = 0;
    for( ; result != 0; ) {
         if( arg.data == NULL ) break;
         memcpy( &string[offset], arg.data, arg.piece_len );
         offset += arg.piece_len;
         string[offset] = '\0';
         if( arg.piece_len == 0 ) break;
         result = api->get_piece( arg_handle, 1, &arg, offset );
    }
    MessageBoxA( NULL, string,
          "SQL Anywhere",
          MB_OK | MB_TASKMODAL );
    free( string );
    return;
}
```

```
short (SQL_CALLBACK *set_value)
(
   void *           arg_handle,
   a_sql_uint32     arg_num,
   an_extfn_value *value
   short            append
);
```

The set_value callback function can be used to set the values of OUT parameters and the RETURNS result of a stored function. Use an arg_num value of 0 to set the RETURNS value. The following is an example.

```
an_extfn_value      retval;

retval.type = DT_LONGVARCHAR;
retval.data = result;
retval.piece_len = retval.len.total_len = (a_sql_uint32) strlen( result );
api->set_value( arg_handle, 0, &retval, 0 );
```


The append argument of set_value determines whether the supplied data replaces (false) or appends to (true) the existing data. You must call set_value with append=FALSE before calling it with append=TRUE for the same argument. The append argument is ignored for fixed length data types.

To return NULL, set the data field of the an_extfn_value structure to NULL.

```
void (SQL_CALLBACK *set_cancel)
(
   void *arg_handle,
   void *cancel_handle
);
```

External functions can get the values of IN or INOUT parameters and set the values of OUT parameters and the RETURNS result of a stored function. There is a case, however, where the parameter values obtained may no longer be valid or the setting of values is no longer necessary. This occurs when an executing SQL statement is canceled. This may occur as the result of an application abruptly disconnecting from the database server. To handle this situation, you can define a special entry point in the library called extfn_cancel. When this function is defined, the server will call it whenever a running SQL statement is canceled.

The extfn_cancel function is called with a handle that can be used in whatever way you deem suitable. A typical use of the handle is to indirectly set a flag to indicate that the calling SQL statement has been canceled.

The value of the handle that is passed can be set by functions in the external library using the set_cancel callback function. This is illustrated by the following code fragment.

```
extern "C" __declspec( dllexport )
void extfn_cancel( void *cancel_handle )
{
    *(short *)cancel_handle = 1;
}

extern "C" __declspec( dllexport )
void mystring( an_extfn_api *api, void *arg_handle )
{
.
.
.
    short  canceled = 0;

    api->set_cancel( arg_handle, &canceled );
.
.
.
    if( canceled )
```

Note that setting a static global "canceled" variable is inappropriate since that would be misinterpreted as all SQL statements on all connections being canceled which is usually not the case. This is why a set_cancel callback function is provided. Make sure to initialize the "canceled" variable before calling set_cancel.

It is important to check the setting of the "canceled" variable at strategic points in your external function. Strategic points would include before and after calling any of the external library call API functions like get_value and set_value. When the variable is set (as a result of extfn_cancel having been called), then the external function can take appropriate termination action. A code fragment based on the earlier example follows:

```
if( canceled )
{
```

```
        free( string );
        return;
    }
```

Notes

The get_piece function for any given argument can only be called immediately after the get_value function for the same argument.

Calling get_value on an OUT parameter returns the type field of the an_extfn_value structure set to the data type of the argument, and returns the data field of the an_extfn_value structure set to NULL.

The header file *extfnapi.h* in the SQL Anywhere installation *SDK\Include* folder contains some additional notes.

The following table shows the conditions under which the functions defined in an_extfn_api return false:

Function	Returns 0 when the following is true; else returns 1
get_value()	• arg_num is invalid; for example, arg_num is greater than the number of arguments for the external function.
get_piece()	• arg_num is invalid; for example, arg_num does not correspond to the argument number used with the previous call to get_value. • The offset is greater than the total length of the value for the arg_num argument. • It is called before get_value has been called.
set_value()	• arg_num is invalid; for example, arg_num is greater than the number of arguments for the external function. • Argument arg_num is input only. • The type of value supplied does not match that of argument arg_num.

Handling data types

Data types

The following SQL data types can be passed to an external library:

SQL data type	sqldef.h	C type
CHAR	DT_FIXCHAR	Character data, with a specified length
VARCHAR	DT_VARCHAR	Character data, with a specified length
LONG VARCHAR, TEXT	DT_LONG-VARCHAR	Character data, with a specified length
UNIQUEIDENTIFIERSTR	DT_FIXCHAR	Character data, with a specified length
XML	DT_LONG-VARCHAR	Character data, with a specified length
NCHAR	DT_NFIX-CHAR	UTF-8 character data, with a specified length
NVARCHAR	DT_NVARCH-AR	UTF-8 character data, with a specified length
LONG NVARCHAR, NTEXT	DT_LONG-NVARCHAR	UTF-8 character data, with a specified length
UNIQUEIDENTIFIER	DT_BINARY	Binary data, 16 bytes long
BINARY	DT_BINARY	Binary data, with a specified length
VARBINARY	DT_BINARY	Binary data, with a specified length
LONG BINARY	DT_LONGBI-NARY	Binary data, with a specified length
TINYINT	DT_TINYINT	1-byte integer
[UNSIGNED] SMALLINT	DT_SMALL-INT, DT_UN-SMALLINT	[Unsigned] 2-byte integer
[UNSIGNED] INT	DT_INT, DT_UNSINT	[Unsigned] 4-byte integer

SQL data type	sqldef.h	C type
[UNSIGNED] BIGINT	DT_BIGINT, DT_UNSBI-GINT	[Unsigned] 8-byte integer
REAL, FLOAT(1-24)	DT_FLOAT	Single precision floating point number
DOUBLE, FLOAT(25-53)	DT_DOUBLE	Double precision floating point number

You cannot use any of the date or time data types, and you cannot use the DECIMAL or NUMERIC data types (including the money types).

To provide values for INOUT or OUT parameters, use the set_value API function. To read IN and INOUT parameters, use the get_value API function.

Determining data types of parameters

After a call to get_value, the type field of the an_extfn_value structure can be used to obtain data type information for the parameter. The following sample code fragment shows how to identify the type of the parameter.

```
an_extfn_value      arg;
a_sql_data_type     data_type;

api->get_value( arg_handle, 1, &arg );
data_type = arg.type & DT_TYPES;
switch( data_type )
{
case DT_FIXCHAR:
case DT_VARCHAR:
case DT_LONGVARCHAR:
    break;
default:
    return;
}
```

For more information on data types, see "Using host variables" on page 522.

UTF-8 types

The UTF-8 data types such as NCHAR, NVARCHAR, LONG NVARCHAR and NTEXT as passed as UTF-8 encoded strings. A function such as the Windows MultiByteToWideChar function can be used to convert a UTF-8 string to a wide-character (Unicode) string.

Passing NULL

You can pass NULL as a valid value for all arguments. Functions in external libraries can supply NULL as a return value for any data type.

Return values

To set a return value in an external function, call the set_value function with an arg_num parameter value of 0. If set_value is not called with arg_num set to 0, the function result is NULL.

It is also important to set the data type of a return value for a stored function call. The following code fragment shows how to set the return data type.

```
an_extfn_value      retval;

retval.type = DT_LONGVARCHAR;
retval.data = result;
retval.piece_len = retval.len.total_len = (a_sql_uint32) strlen( result );
api->set_value( arg_handle, 0, &retval, 0 );
```

Unloading external libraries

The system procedure, dbo.sa_external_library_unload, can be used to unload an external library when the library is not in use. The procedure takes one optional parameter, a long varchar. The parameter specifies the name of the library to be unloaded. If no parameter is specified, all external libraries not in use will be unloaded.

The following example unloads an external function library.

```
call sa_external_library_unload('library.dll')
```

This function is useful when developing a set of external functions because you do not have to shut down the database server to install a newer version of the library.

CHAPTER 16

SQL Anywhere Perl DBD::SQLAnywhere API

Contents

Introduction to DBD::SQLAnywhere

The DBD::SQLAnywhere interface provides access to SQL Anywhere databases from scripts written in Perl. DBD::SQLAnywhere is a driver for the Database Independent Interface for Perl (DBI) module written by Tim Bunce. Once you have installed the DBI module and DBD::SQLAnywhere, you can access and change the information in SQL Anywhere databases from Perl.

The DBD::SQLAnywhere driver is thread-safe when using Perl with ithreads.

Requirements

The DBD::SQLAnywhere interface requires the following components.

- Perl 5.6.0 or newer. On Windows, ActivePerl 5.6.0 build 616 or later is required.
- DBI 1.34 or newer.
- A C compiler. On Windows, only the Microsoft Visual C++ compiler is supported.

The following sections provide assistance with installing Perl, DBI, and the DBD::SQLAnywhere driver software.

Installing DBD::SQLAnywhere on Windows

To prepare your computer

1. Install ActivePerl 5.6.0 or later. You can use the ActivePerl installer to install Perl and configure your computer. You do not need to recompile Perl.
2. Install Microsoft Visual Studio and configure your environment.

   If you did not choose to configure your environment at install time, you must set your PATH, LIB, and INCLUDE environment variables correctly before proceeding. Microsoft provides a batch file for this purpose. For example, a batch file called *vcvars32.bat* is included in the *vc\bin* subdirectory of the Visual Studio 2005 or 2008 installation. Open a new system command prompt and run this batch file before continuing.

To install the DBI Perl module on Windows

1. At a command prompt, change to the *bin* subdirectory of your ActivePerl installation directory.

   The system command prompt is strongly recommended as the following steps may not work from alternative shells.
2. Using the Perl Module Manager, enter the following command.

   ```
   ppm query dbi
   ```

   If ppm fails to run, check that Perl is installed correctly.

   This command should generate two lines of text similar to those shown below. In this case, the information indicates that ActivePerl version 5.8.1 build 807 is running and that DBI version 1.38 is installed.

   ```
   Querying target 1 (ActivePerl 5.8.1.807)
     1. DBI [1.38] Database independent interface for Perl
   ```

   Later versions of Perl may show instead a table similar to the following. In this case, the information indicates that DBI version 1.58 is installed.

name	version	abstract	area
DBI	1.58	Database independent interface for Perl	perl

   If DBI is not installed, you must install it. To do so, enter the following command at the ppm prompt.

   ```
   ppm install dbi
   ```

To install DBD::SQLAnywhere on Windows

1. At a system command prompt, change to the *SDK\Perl* subdirectory of your SQL Anywhere installation.
2. Enter the following commands to build and test DBD::SQLAnywhere.

   ```
   perl Makefile.PL
   ```

   ```
   nmake
   ```


If for any reason you need to start over, you can run the command **nmake clean** to remove any partially built targets.

3. To test DBD::SQLAnywhere, copy the sample database file to your *SDK\Perl* directory and make the tests.

   ```
   copy "samples-dir\demo.db" .
   ```

   For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

   ```
   dbeng11 demo
   ```

   ```
   nmake test
   ```

   If the tests do not run, ensure that the *bin32* or *bin64* subdirectory of the SQL Anywhere installation is in your path.

4. To complete the installation, execute the following command at the same prompt.

   ```
   nmake install
   ```

The DBD::SQLAnywhere interface is now ready to use.

Installing DBD::SQLAnywhere on Unix

The following procedure documents how to install the DBD::SQLAnywhere interface on the supported Unix platforms, including Mac OS X.

To prepare your computer

1. Install ActivePerl 5.6.0 build 616 or later.
2. Install a C compiler.

To install the DBI Perl module on Unix

1. Download the DBI module source from www.cpan.org.
2. Extract the contents of this file into a new directory.
3. At a command prompt, change to the new directory and execute the following commands to build the DBI module.

   ```
   perl Makefile.PL
   make
   ```

   If for any reason you need to start over, you can use the command **make clean** to remove any partially built targets.
4. Use the following command to test the DBI module.

   ```
   make test
   ```
5. To complete the installation, execute the following command at the same prompt.

   ```
   make install
   ```
6. Optionally, you can now delete the DBI source tree. It is no longer required.

To install DBD::SQLAnywhere on Unix

1. Make sure the environment is set up for SQL Anywhere.

   Depending on which shell you are using, enter the appropriate command to source the SQL Anywhere configuration script from the SQL Anywhere installation directory:

In this shell ...	... use this command
sh, ksh, or bash	. *bin/sa_config.sh*
csh or tcsh	source *bin/sa_config.csh*

2. At a shell prompt, change to the *sdk/perl* subdirectory of your SQL Anywhere installation.
3. At a system command prompt, enter the following commands to build DBD::SQLAnywhere.

   ```
   perl Makefile.PL
   make
   ```

If for any reason you need to start over, you can use the command **make clean** to remove any partially built targets.

4. To test DBD::SQLAnywhere, copy the sample database file to your *sdk/perl* directory and make the tests.

   ```
   cp samples-dir/demo.db .
   dbeng11 demo
   make test
   ```

   If the tests do not run, ensure that the *bin32* or *bin64* subdirectory of the SQL Anywhere installation is in your path.

5. To complete the installation, execute the following command at the same prompt.

   ```
   make install
   ```

The DBD::SQLAnywhere interface is now ready to use.

Writing Perl scripts that use DBD::SQLAnywhere

This section provides an overview of how to write Perl scripts that use the DBD::SQLAnywhere interface. DBD::SQLAnywhere is a driver for the DBI module. Complete documentation for the DBI module is available online at dbi.perl.org.

Loading the DBI module

To use the DBD::SQLAnywhere interface from a Perl script, you must first tell Perl that you plan to use the DBI module. To do so, include the following line at the top of the file.

```
use DBI;
```

In addition, it is highly recommended that you run Perl in strict mode. This statement, which for example makes explicit variable definitions mandatory, is likely to greatly reduce the chance that you will run into mysterious errors due to such common mistakes as typographical errors.

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
```

The DBI module automatically loads the DBD drivers, including DBD::SQLAnywhere, as required.

Opening and closing a connection

Generally, you open a single connection to a database and then perform all the required operations through it by executing a sequence of SQL statements. To open a connection, you use the connect method. The return value is a handle to the database connection that you use to perform subsequent operations on that connection.

The parameters to the connect method are as follows:

1. "DBI:SQLAnywhere:" and additional connection parameters separated by semicolons.
2. A user name. Unless this string is blank, ";UID=*value*" is appended to the connection string.
3. A password value. Unless this string is blank, ";PWD=*value*" is appended to the connection string.
4. A pointer to a hash of default values. Settings such as AutoCommit, RaiseError, and PrintError may be set in this manner.

The following code sample opens and closes a connection to the SQL Anywhere sample database. You must start the database server and sample database before running this script.

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "demo";
my $data_src = "DBI:SQLAnywhere:ENG=$database;DBN=$database";
my $uid      = "DBA";
my $pwd      = "sql";
```

```
my %defaults = (
     AutoCommit => 1, # Autocommit enabled.
     PrintError => 0  # Errors not automatically printed.
   );
my $dbh = DBI->connect($data_src, $uid, $pwd, \%defaults)
  or die "Cannot connect to $data_src: $DBI::errstr\n";
$dbh->disconnect;
exit(0);
__END__
```

Optionally, you can append the user name or password value to the data-source string instead of supplying them as separate parameters. If you do so, supply a blank string for the corresponding argument. For example, in the above script may be altered by replacing the statement that opens the connections with these statements:

```
$data_src .= ";UID=$uid";
$data_src .= ";PWD=$pwd";
my $dbh = DBI->connect($data_src, '', '', \%defaults)
  or die "Can't connect to $data_source: $DBI::errstr\n";
```

Selecting data

Once you have obtained a handle to an open connection, you can access and modify data stored in the database. Perhaps the simplest operation is to retrieve some rows and print them out.

SQL statements that return row sets must be prepared before being executed. The prepare method returns a handle to the statement. You use the handle to execute the statement, then retrieve meta information about the result set and the rows of the result set.

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "demo";
my $data_src = "DBI:SQLAnywhere:ENG=$database;DBN=$database";
my $uid      = "DBA";
my $pwd      = "sql";
my $sel_stmt = "SELECT ID, GivenName, Surname
                FROM Customers
                ORDER BY GivenName, Surname";
my %defaults = (
     AutoCommit => 0, # Require explicit commit or rollback.
     PrintError => 0
   );
my $dbh = DBI->connect($data_src, $uid, $pwd, \%defaults)
  or die "Can't connect to $data_src: $DBI::errstr\n";
&db_query($sel_stmt, $dbh);
$dbh->rollback;
$dbh->disconnect;
exit(0);

sub db_query {
      my($sel, $dbh) = @_;
      my($row, $sth) = undef;
      $sth = $dbh->prepare($sel);
      $sth->execute;
      print "Fields:     $sth->{NUM_OF_FIELDS}\n";
      print "Params:     $sth->{NUM_OF_PARAMS}\n\n";
      print join("\t\t", @{$sth->{NAME}}), "\n\n";
      while($row = $sth->fetchrow_arrayref) {
```



```
            print join("\t\t", @$row), "\n";
        }
        $sth = undef;
}
__END__
```

Prepared statements are not dropped from the database server until the Perl statement handle is destroyed. To destroy a statement handle, reuse the variable or set it to undef. Calling the finish method does not drop the handle. In fact, the finish method should not be called, except when you have decided not to finish reading a result set.

To detect handle leaks, the SQL Anywhere database server limits the number of cursors and prepared statements permitted to a maximum of 50 per connection by default. The resource governor automatically generates an error if these limits are exceeded. If you get this error, check for undestroyed statement handles. Use prepare_cached sparingly, as the statement handles are not destroyed.

If necessary, you can alter these limits by setting the max_cursor_count and max_statement_count options. See "max_cursor_count option [database]" [*SQL Anywhere Server - Database Administration*], and "max_statement_count option [database]" [*SQL Anywhere Server - Database Administration*].

Inserting rows

Inserting rows requires a handle to an open connection. The simplest method is to use a parameterized INSERT statement, meaning that question marks are used as place holders for values. The statement is first prepared, and then executed once per new row. The new row values are supplied as parameters to the execute method.

The following sample program inserts two new customers. Although the row values appear as literal strings, you may want to read the values from a file.

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "demo";
my $data_src = "DBI:SQLAnywhere:ENG=$database;DBN=$database";
my $uid      = "DBA";
my $pwd      = "sql";
my $ins_stmt = "INSERT INTO Customers (ID, GivenName, Surname,
                        Street, City, State, Country, PostalCode,
                        Phone, CompanyName)
                VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";
my %defaults = (
     AutoCommit => 0, # Require explicit commit or rollback.
     PrintError => 0
   );
my $dbh = DBI->connect($data_src, $uid, $pwd, \%defaults)
     or die "Can't connect to $data_src: $DBI::errstr\n";
&db_insert($ins_stmt, $dbh);
$dbh->commit;
$dbh->disconnect;
exit(0);

sub db_insert {
      my($ins, $dbh) = @_;
      my($sth) = undef;
```

```
        my @rows = (
          "801,Alex,Alt,5 Blue Ave,New York,NY,USA,10012,5185553434,BXM",
          "802,Zach,Zed,82 Fair St,New York,NY,USA,10033,5185552234,Zap"
         );
        $sth = $dbh->prepare($ins);
        my $row = undef;
        foreach $row ( @rows ) {
          my @values = split(/,/, $row);
          $sth->execute(@values);
        }
}
__END__
```

CHAPTER 17

SQL Anywhere for Python Database API

Contents

Introduction to sqlanydb

The sqlanydb interface provides access to SQL Anywhere databases from scripts written in Python. The sqlanydb module implements, with extensions, the Python Database API specification v2.0 written by Marc-André Lemburg. Once you have installed the sqlanydb module, you can access and change the information in SQL Anywhere databases from Python.

For information about the Python Database API specification v2.0, visit Python Database API specification v2.0.

The sqlanydb module is thread-safe when using Python with threads.

Requirements

The sqlanydb module requires the following components.

- Python 2.4 or newer (2.5 or newer is recommended).
- The ctypes module is required. If ctypes is not included in your Python installation, install it. Installs can be found in the SourceForge.net files section at http://sourceforge.net/project/showfiles.php?group_id=71702, or can be automatically downloaded and installed with Peak EasyInstall. To download Peak EasyInstall, visit http://peak.telecommunity.com/DevCenter/EasyInstall.

The following sections provide assistance with installing Python and the sqlanydb module.

Installing sqlanydb on Windows

To prepare your computer

1. Install Python 2.4 or later.
2. Install the ctypes module if missing.

To install the sqlanydb module on Windows

1. At a system command prompt, change to the *SDK\Python* subdirectory of your SQL Anywhere installation.
2. Run the following command to install sqlanydb.

   ```
   python setup.py install
   ```

3. To test sqlanydb, copy the sample database file to your *SDK\Python* directory and run a test.

   ```
   copy "samples-dir\demo.db" .
   dbeng11 demo
   python Scripts\test.py
   ```

   If the tests do not run, ensure that the *bin32* or *bin64* subdirectory of the SQL Anywhere installation is in your path.

The sqlanydb module is now ready to use.

Installing sqlanydb on Unix

The following procedure documents how to install the sqlanydb module on the supported Unix platforms, including Mac OS X.

To prepare your computer

1. Install Python 2.4 or later.
2. Install the ctypes module if missing.

To install the sqlanydb module on Unix

1. Make sure the environment is set up for SQL Anywhere.

   Depending on which shell you are using, enter the appropriate command to source the SQL Anywhere configuration script from the SQL Anywhere installation directory:

In this shell ...	... use this command
sh, ksh, or bash	. *bin/sa_config.sh*
csh or tcsh	source *bin/sa_config.csh*

2. At a shell prompt, change to the *sdk/python* subdirectory of your SQL Anywhere installation.
3. Enter the following command to install sqlanydb.

   ```
   python setup.py install
   ```

4. To test sqlanydb, copy the sample database file to your *sdk/python* directory and run a test.

   ```
   cp samples-dir/demo.db .
   dbeng11 demo
   python scripts/test.py
   ```

   If the tests do not run, ensure that the *bin32* or *bin64* subdirectory of the SQL Anywhere installation is in your path.

The sqlanydb module is now ready to use.

Writing Python scripts that use sqlanydb

This section provides an overview of how to write Python scripts that use the sqlanydb interface. Complete documentation for the API is available online at Python Database API specification v2.0.

Loading the sqlanydb module

To use the sqlanydb module from a Python script, you must first load it by including the following line at the top of the file.

```
import sqlanydb
```

Opening and closing a connection

Generally, you open a single connection to a database and then perform all the required operations through it by executing a sequence of SQL statements. To open a connection, you use the connect method. The return value is a handle to the database connection that you use to perform subsequent operations on that connection.

The parameters to the connect method are specified as a series of keyword=value pairs delimited by commas.

```
sqlanydb.connect( keyword=value, ...)
```

Some common connection parameters are as follows:

- **DataSourceName="*dsn*"** A short form for this connection parameter is **DSN**="*dsn*". An example is DataSourceName="SQL Anywhere 11 Demo".
- **UserID="*user-id*"** A short form for this connection parameter is **UID**="*user-id*". An example is UserID="DBA".
- **Password="*passwd*"** A short form for this connection parameter is **PWD**="*passwd*". An example is Password="sql".
- **DatabaseFile="*db-file*"** A short form for this connection parameter is **DBF**="*db-file*". An example is DatabaseFile="demo.db"

For the complete list of connection parameters, see "Connection parameters and network protocol options" [*SQL Anywhere Server - Database Administration*].

The following code sample opens and closes a connection to the SQL Anywhere sample database. You must start the database server and sample database before running this script.

```
import sqlanydb

# Create a connection object
con = sqlanydb.connect( userid="DBA",
                        password="sql" )
# Close the connection
con.close()
```

To avoid starting the database server manually, you could use a data source that is configured to start the server. This is shown in the following example.

```
import sqlanydb

# Create a connection object
con = sqlanydb.connect( DSN="SQL Anywhere 11 Demo" )

# Close the connection
con.close()
```

Selecting data

Once you have obtained a handle to an open connection, you can access and modify data stored in the database. Perhaps the simplest operation is to retrieve some rows and print them out.

The cursor method is used to create a cursor on the open connection. The execute method is used to create a result set. The fetchall method is used to obtain the rows in this result set.

```
import sqlanydb

# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid="DBA",
                        password="sql" )
cursor = con.cursor()

# Execute a SQL string
sql = "SELECT * FROM Employees"
cursor.execute(sql)

# Get a cursor description which contains column names
desc = cursor.description
print len(desc)

# Fetch all results from the cursor into a sequence,
# display the values as column name=value pairs,
# and then close the connection
rowset = cursor.fetchall()
for row in rowset:
    for col in range(len(desc)):
        print "%s=%s" % (desc[col][0], row[col] )
    print
cursor.close()
con.close()
```

Inserting rows

The simplest way to insert rows into a table is to use a non-parameterized INSERT statement, meaning that values are specified as part of the SQL statement. A new statement is constructed and executed for each new row. As in the previous example, a cursor is required to execute SQL statements.

The following sample program inserts two new customers into the sample database. Before disconnecting, it commits the transactions to the database.

```
import sqlanydb

# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid="DBA", pwd="sql" )
```



```
cursor = con.cursor()
cursor.execute("DELETE FROM Customers WHERE ID > 800")

rows = ((801,'Alex','Alt','5 Blue Ave','New York','NY',
        'USA','10012','5185553434','BXM'),
        (802,'Zach','Zed','82 Fair St','New York','NY',
        'USA','10033','5185552234','Zap'))

# Set up a SQL INSERT
parms = ("'%s'," * len(rows[0]))[:-1]
sql = "INSERT INTO Customers VALUES (%s)" % (parms)
print sql % rows[0]
cursor.execute(sql % rows[0])
print sql % rows[1]
cursor.execute(sql % rows[1])
cursor.close()
con.commit()
con.close()
```

An alternate technique is to use a parameterized INSERT statement, meaning that question marks are used as place holders for values. The executemany method is used to execute an INSERT statement for each member of the set of rows. The new row values are supplied as a single argument to the executemany method.

```
import sqlanydb

# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid="DBA", pwd="sql" )
cursor = con.cursor()
cursor.execute("DELETE FROM Customers WHERE ID > 800")

rows = ((801,'Alex','Alt','5 Blue Ave','New York','NY',
        'USA','10012','5185553434','BXM'),
        (802,'Zach','Zed','82 Fair St','New York','NY',
        'USA','10033','5185552234','Zap'))

# Set up a parameterized SQL INSERT
parms = ("?," * len(rows[0]))[:-1]
sql = "INSERT INTO Customers VALUES (%s)" % (parms)
print sql
cursor.executemany(sql, rows)
cursor.close()
con.commit()
con.close()
```

Although both examples may appear to be equally suitable techniques for inserting row data into a table, the latter example is superior for a couple of reasons. If the data values are obtained by prompts for input, then the first example is susceptible to injection of rogue data including SQL statements. In the first example, the execute method is called for each row to be inserted into the table. In the second example, the executemany method is called only once to insert all the rows into the table.

CHAPTER 18

SQL Anywhere PHP API

Contents

Introduction to the SQL Anywhere PHP module

PHP, which stands for Hypertext Preprocessor, is an open source scripting language. Although it can be used as a general-purpose scripting language, it was designed to be a convenient language in which to write scripts that could be embedded with HTML documents. Unlike scripts written in JavaScript, which are frequently executed by the client, PHP scripts are processed by the web server, and the resulting HTML output sent to the clients. The syntax of PHP is derived from that of other popular languages, such as Java and Perl.

To make it a convenient language in which to develop dynamic web pages, PHP provides the ability to retrieve information from many popular databases, such as SQL Anywhere. Included with SQL Anywhere are two modules that provide access to SQL Anywhere databases from PHP. You can use these modules and the PHP language to write stand-alone scripts and create dynamic web pages that rely on information stored in SQL Anywhere databases.

Prebuilt versions of the PHP modules are provided for Windows, Linux, and Solaris and are installed in the operating-system-specific binaries subdirectories of your SQL Anywhere installation. Source code for the SQLAnywhere PHP module is installed in the *SDK\PHP* subdirectory of your SQL Anywhere installation.

Requirements

Before you can use the SQL Anywhere PHP module, you must also install the following components:

- The PHP 5 binaries for your platform, which can be downloaded from http://www.php.net.
- A web server, if you want to run PHP scripts within a web server. SQL Anywhere can be run on the same computer as the web server, or on a different computer.
- For Windows, the SQL Anywhere client software *dblib11.dll* and *dbcapi.dll*.
- For Linux/Unix, the SQL Anywhere client software *libdblib11.so* and *libdbcapi.so*.
- For Mac OS X, the SQL Anywhere client software *libdblib11.dylib* and *libdbapi.dylib*.

The following sections provide assistance with installing the SQL Anywhere PHP module.

Installing and configuring SQL Anywhere PHP

The following sections describe how to install and configure the SQL Anywhere PHP module.

Choosing which PHP module to use

On Windows, SQL Anywhere includes modules for PHP version 5. The module file names for the supported PHP versions follow this pattern:

```
php-5.x.y_sqlanywhere.dll
```

On Linux and Solaris, SQL Anywhere includes both 64-bit and 32-bit versions of the modules for PHP version 5. It also includes both threaded and non-threaded modules. If you are using the CGI version of PHP or if you are using Apache 1.x, use the non-threaded module. If you are using Apache 2.x, use the threaded module. The module file names for the supported PHP versions follow this pattern:

```
php-5.x.y_sqlanywhere[_r].so
```

The "5.x.y" represents the PHP version (for example, 5.2.6). For Linux and Solaris, the threaded version of the PHP module has *_r* appended to the file name. Windows versions are implemented as Dynamic Link Libraries and Linux versions are implemented as Shared Objects.

Installing the PHP module on Windows

To use the SQL Anywhere PHP module on Windows, you must copy the DLL from the SQL Anywhere installation directory and add it to your PHP installation. Optionally, you can add an entry to your PHP initialization file to load the module, so you do not need to load it manually in each script.

To install the PHP module on Windows

1. Locate the *php.ini* file for your PHP installation, and open it in a text editor. Locate the line that specifies the location of the **extension_dir** directory. If **extension_dir** is not set to any specific directory, it is a good idea to set it to point to an isolated directory for better system security.
2. Copy the file *php-5.x.y_sqlanywhere.dll*, where *5.x.y* is the PHP version number from the *Bin32* subdirectory of your SQL Anywhere installation, to the directory specified by the **extension_dir** entry in the *php.ini* file.

   Note
   If your version of PHP is more recent than the SQL Anywhere PHP modules provided by SQL Anywhere, try using the most recent module provided. For example, if you installed PHP 5.2.7, and the most recent SQL Anywhere PHP module is *php-5.2.6_sqlanywhere.dll*, use *php-5.2.6_sqlanywhere.dll*.
3. Add the following line to the Dynamic Extensions section of the *php.ini* file to load the SQL Anywhere PHP driver automatically.

   ```
   extension=php-5.x.y_sqlanywhere.dll
   ```

where *5.x.y* reflects the version number of the SQL Anywhere PHP module copied in the previous step.

An alternative to automatically loading the PHP driver is to load it manually in each script that requires it. See "Configuring the SQL Anywhere PHP module" on page 630.

4. Make sure that the *Bin32* subdirectory of your SQL Anywhere installation is in your path. The SQL Anywhere PHP extension DLL requires the *Bin32* directory to be in your path.
5. At a command prompt, enter the following command to start the SQL Anywhere sample database.

   ```
   dbeng11 samples-dir\demo.db
   ```

   The command starts a database server using the sample database.
6. At a command prompt, change to the *SDK\PHP\Examples* subdirectory of your SQL Anywhere installation, and enter the following command:

   ```
   php test.php
   ```

   Messages similar to the following should appear. If the PHP command is not recognized, verify that PHP is in your path.

   ```
   Installation successful
   Using php-5.2.6_sqlanywhere.dll
   Connected successfully
   ```

   If the SQL Anywhere PHP driver does not load, you can use the command "php -i" for helpful information about your PHP setup. Search for **extension_dir** and **sqlanywhere** in the output from this command.
7. When you are done, stop the SQL Anywhere database server by clicking Shut Down in the database server messages window.

For more information, see "Creating PHP test pages" on page 632.

Installing the PHP module on Linux/Solaris

To use the SQL Anywhere PHP module on Linux or Solaris, you must copy the shared object from the SQL Anywhere installation directory and add it to your PHP installation. Optionally, you can add an entry to your PHP initialization file, *php.ini*, to load the module, so you do not need to load it manually in each script.

To install the PHP module on Linux/Solaris

1. Locate the *php.ini* file of your PHP installation, and open it in a text editor. Locate the line that specifies the location of the **extension_dir** directory. If **extension_dir** is not set to any specific directory, it is a good idea to set it to point to an isolated directory for better system security.
2. Copy the shared object from the *lib32* or *lib64* subdirectory of your SQL Anywhere installation to the directory specified by the **extension_dir** entry in the *php.ini* file. Your choice of shared object will depend on the version of PHP that you have installed and whether it is a 32-bit or a 64-bit version.

> **Note**
> If your version of PHP is more recent than the shared object provided by SQL Anywhere, try using the most recent shared object provided. For example, if you installed PHP 5.2.7, and the most recent shared object is *php-5.2.6_sqlanywhere_r.so*, use *php-5.2.6_sqlanywhere_r.so*.

For information about which version of the shared object to use, see "Choosing which PHP module to use" on page 627.

3. Optionally, add the following line to the *php.ini* file to load the SQL Anywhere PHP driver automatically. Alternatively, you can load it manually with a few extra lines of code at the start of each script that requires it. The entry must identify the shared object you copied, which is either

   ```
   extension=php-5.x.y_sqlanywhere.so
   ```

   or, for the thread-safe shared object,

   ```
   extension=php-5.x.y_sqlanywhere_r.so
   ```

   where *5.x.y* is the version number of the PHP shared object copied in the previous step.

4. Before attempting to use the PHP module, verify that your environment is set up for SQL Anywhere. Depending on which shell you are using, enter the appropriate command to source the SQL Anywhere configuration script from the SQL Anywhere installation directory:

In this shell ...	... use this command
sh, ksh, or bash	. /bin32/sa_config.sh
csh or tcsh	source /bin32/sa_config.csh

   The 32-bit version of the SQL Anywhere PHP extension DLL requires the *bin32* directory to be in your path. The 64-bit version of the SQL Anywhere PHP extension DLL requires the *bin64* directory to be in your path.

5. At a command prompt, enter the following command to start the SQL Anywhere sample database:

   ```
   dbeng11 samples-dir/demo.db
   ```

6. At a command prompt, change to the *sdk/php/examples* subdirectory of your SQL Anywhere installation. Enter the following command:

   ```
   php test.php
   ```

   Messages similar to the following should appear. If the php command is not recognized, verify that php is in your path.

   ```
   Installation successful
   Using php-5.2.6_sqlanywhere.so
   Connected successfully
   ```

7. When you are done, stop the database server.

For more information, see "Creating PHP test pages" on page 632.

Building the PHP module on Unix and Mac OS X

To use the SQL Anywhere PHP module on other versions of Unix or Mac OS X, you must build the PHP module from the source code which is installed in the *sdk/php* subdirectory of your SQL Anywhere

installation. For more information, see Serving Content from SQL Anywhere Databases Using Apache and PHP.

Configuring the SQL Anywhere PHP module

The behavior of the SQL Anywhere PHP driver can be controlled by setting values in the PHP initialization file, *php.ini*. The following entries are supported:

- **extension** Causes PHP to load the SQL Anywhere PHP module automatically each time PHP starts. Adding this entry to your PHP initialization file is optional, but if you don't add it, each script you write must start with a few lines of code that ensure that this module is loaded. The following entry is used for Windows platforms.

  ```
  extension=php-5.x.y_sqlanywhere.dll
  ```

  On Linux platforms, use one of the following entries. The second entry is thread safe.

  ```
  extension=php-5.x.y_sqlanywhere.so
  ```

  ```
  extension=php-5.x.y_sqlanywhere_r.so
  ```

  In these entries, *5.x.y* identifies the PHP version.

  If the SQL Anywhere module is not always automatically loaded when PHP starts, you must prefix each script you write with the following lines of code. This code ensures that the SQL Anywhere PHP module is loaded.

  ```
  # Ensure that the SQL Anywhere PHP module is loaded
  if( !extension_loaded('sqlanywhere') ) {
      # Find out which version of PHP is running
      $version = phpversion();
      $module_name = 'php-'.$version.'_sqlanywhere';
      if( strtoupper(substr(PHP_OS, 0, 3) == 'WIN' )) {
          $module_ext = '.dll';
      } else {
          $module_ext = '.so';
      }
      dl( $module_name.$module_ext );
  }
  ```

- **allow_persistent** Allows persistent connections when set to 1. It does not allow them when set to 0. The default value is 1.

  ```
  sqlanywhere.allow_persistent=1
  ```

- **max_persistent** Sets the maximum number of persistent connections. The default value is -1, which means no limit.

  ```
  sqlanywhere.max_persistent=-1
  ```

- **max_connections** Sets the maximum number of connections that can be opened at once through the SQL Anywhere PHP module. The default value is -1, which means no limit.

  ```
  sqlanywhere.max_connections=-1
  ```

- **auto_commit** Specifies whether the database server performs a commit operation automatically. The commit is performed immediately following the execution of each statement when set to 1. When set to

0, transactions should be ended manually with either the sasql_commit or sasql_rollback functions, as appropriate. The default value is 1.

```
sqlanywhere.auto_commit=1
```

- **row_counts** Returns the exact number of rows affected by an operation when set to 1 or an estimate when set to 0. The default value is 0.

  ```
  sqlanywhere.row_counts=0
  ```

- **verbose_errors** Returns verbose errors and warnings when set to 1. Otherwise, you must call the sasql_error or sasql_errorcode functions to get further error information. The default value is 1.

  ```
  sqlanywhere.verbose_errors=1
  ```

For more information, see "sasql_set_option" on page 655.

Running PHP test scripts in your web pages

This section describes how to write PHP test scripts that query the sample database and display information about PHP.

Creating PHP test pages

The following instructions apply to all configurations.

To test whether PHP is set up properly, the following procedure describes how to create and run a web page that calls phpinfo. phpinfo is a PHP function that generates a page of system setup information. The output tells you whether PHP is working properly.

For information about installing PHP, see http://us2.php.net/install.

To create a PHP information test page

1. Create a file in your root web content directory named *info.php*.

   If you are not sure which directory to use, check your web server's configuration file. In Apache installations, the content directory is often called *htdocs*. If you are using Mac OS X, the web content directory name may depend on which account you are using:

   - If you are the System Administrator on a Mac OS X system, use */Library/WebServer/Documents*.
   - If you are a Mac OS X user, place the file in */Users/your-user-name/Sites/*.

2. Insert the following code into this file:

   ```
   <?php phpinfo() ?>
   ```

   Alternatively, once PHP is properly installed and configured, you can also create a test web page by issuing the following command at a command prompt.

   ```
   php -I > info.html
   ```

   This confirms that your installation of PHP and your web server are working together properly.

3. At a command prompt, enter the following command to start the SQL Anywhere sample database (if you have not already done so):

   ```
   dbeng11 samples-dir\demo.db
   ```

4. To test that PHP and your web server are working correctly with SQL Anywhere:

   a. Copy the file *connect.php* from your PHP examples directory to your root web content directory.

   b. From a web browser, access the *connect.php* page.

      The message `Connected successfully` should appear.

To create the query page that uses the SQL Anywhere PHP module

1. Create a file containing the following PHP code in your root web content directory named *sa_test.php*.
2. Insert the following PHP code into this file:


```
<?php
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  $result = sasql_query( $conn, "SELECT * FROM Employees" );
  sasql_result_all( $result );
  sasql_free_result( $result );
  sasql_disconnect( $conn );
?>
```

Accessing your test web pages

The following procedure describes how to view your test pages from a web browser, after installing and configuring PHP and the SQL Anywhere PHP module.

To view your web pages

1. Restart your web server.

   For example, to start the Apache web server, run the following command from the *bin* subdirectory of your Apache installation:

   ```
   apachectl start
   ```

2. On Linux or Mac OS X, set the SQL Anywhere environment variables using one of the supplied scripts.

   Depending on which shell you are using, enter the appropriate command to source the SQL Anywhere configuration script from your SQL Anywhere installation directory:

In this shell ...	... use this command
sh, ksh, or bash	`. /bin32/sa_config.sh`
csh or tcsh	`source /bin32/sa_config.csh`

3. Start the SQL Anywhere database server.

   For example, to access the test web pages described above, use the following command to start the SQL Anywhere sample database.

   ```
   dbeng11 samples-dir\demo.db
   ```

4. To access the test pages from a browser that is running on the same computer as the server, enter the following URLs:

For this test page ...	... use this URL
info.php	`http://localhost/info.php`
sa_test.php	`http://localhost/sa_test.php`

   If everything is configured correctly, the sa_test page displays the contents of the Employees table.

Writing PHP scripts

This section describes how to write PHP scripts that use the SQL Anywhere PHP module to access SQL Anywhere databases.

The source code for these examples, as well as others, is located in the *SDK\PHP\Examples* subdirectory of your SQL Anywhere installation.

Connecting to a database

To make a connection to a database, pass a standard SQL Anywhere connection string to the database server as a parameter to the sasql_connect function. The <?php and ?> tags tell the web server that it should let PHP execute the code that lies between them and replace it with the PHP output.

The source code for this example is contained in your SQL Anywhere installation in a file called *connect.php*.

```
<?php
  # Connect using the default user ID and password
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      echo "Connection failed\n";
  } else {
      echo "Connected successfully\n";
      sasql_close( $conn );
  }?>
```

The first block of code verifies that the PHP module is loaded. If you added the line to your PHP initialization file to load it automatically, this block of code is unnecessary. If you did not configure PHP to automatically load the SQL Anywhere PHP module at start time, you must add this code to the other sample scripts.

The second block attempts to make a connection. For this code to succeed, the SQL Anywhere sample database must be running.

Retrieving data from a database

One use of PHP scripts in web pages is to retrieve and display information contained in a database. The following examples demonstrate some useful techniques.

Simple select query

The following PHP code demonstrates a convenient way to include the result set of a SELECT statement in a web page. This sample is designed to connect to the SQL Anywhere sample database and return a list of customers.

This code can be embedded in a web page, provided your web server is configured to execute PHP scripts.

The source code for this sample is contained in your SQL Anywhere installation in a file called *query.php*.

```
<?php
  # Connect using the default user ID and password
```



```
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      echo "sasql_connect failed\n";
  } else {
      echo "Connected successfully\n";
      # Execute a SELECT statement
      $result = sasql_query( $conn, "SELECT * FROM Customers" );
      if( ! $result ) {
          echo "sasql_query failed!";
      } else {
          echo "query completed successfully\n";
          # Generate HTML from the result set
          sasql_result_all( $result );
          sasql_free_result( $result );
      }
      sasql_close( $conn );
  }
?>
```

The sasql_result_all function fetches all the rows of the result set and generates an HTML output table to display them. The sasql_free_result function releases the resources used to store the result set.

Fetching by column name

In certain cases, you may not want to display all the data from a result set, or you may want to display the data in a different manner. The following sample illustrates how you can exercise greater control over the output format of the result set. PHP allows you to display as much information as you want in whatever manner you choose.

The source code for this sample is contained in your SQL Anywhere installation in a file called *fetch.php*.

```
<?php
  # Connect using the default user ID and password
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      die ("Connection failed");
  } else {
      # Connected successfully.
  }
  # Execute a SELECT statement
  $result = sasql_query( $conn, "SELECT * FROM Customers" );
  if( ! $result ) {
      echo "sasql_query failed!";
      return 0;
  } else {
      echo "query completed successfully\n";
  }
  # Retrieve meta information about the results
  $num_cols = sasql_num_fields( $result );
  $num_rows = sasql_num_rows( $result );
  echo "Num of rows = $num_rows\n";
  echo "Num of cols = $num_cols\n";
  while( ($field = sasql_fetch_field( $result )) ) {
      echo "Field # : $field->id \n";
      echo "\tname   : $field->name \n";
      echo "\tlength : $field->length \n";
      echo "\ttype   : $field->type \n";
  }
  # Fetch all the rows
  $curr_row = 0;
  while( ($row = sasql_fetch_row( $result )) ) {
      $curr_row++;
```

```
        $curr_col = 0;
        while( $curr_col < $num_cols ) {
            echo "$row[$curr_col]\t|";
            $curr_col++;
        }
        echo "\n";
    }
    # Clean up.
    sasql_free_result( $result );
    sasql_disconnect( $conn );
?>
```

The sasql_fetch_array function returns a single row from the table. The data can be retrieved by column names and column indexes.

The sasql_fetch_assoc function returns a single row from the table as an associative array. The data can be retrieved by using the column names as indexes. The following is an example.

```
<?php
  # Connect using the default user ID and password
  $conn = sasql_connect("UID=DBA;PWD=sql");

  /* check connection */
  if( sasql_errorcode() ) {
      printf("Connect failed: %s\n", sasql_error());
      exit();
  }

  $query = "SELECT Surname, Phone FROM Employees ORDER by EmployeeID";

  if( $result = sasql_query($conn, $query) ) {

      /* fetch associative array */
      while( $row = sasql_fetch_assoc($result) ) {
          printf ("%s (%s)\n", $row["Surname"], $row["Phone"]);
      }

      /* free result set */
      sasql_free_result($result);
  }

  /* close connection */
  sasql_close($conn);
?>
```

Two other similar methods are provided in the PHP interface: sasql_fetch_row returns a row that can be searched by column indexes only, while sasql_fetch_object returns a row that can be searched by column names only.

For an example of the sasql_fetch_object function, see the *fetch_object.php* example script.

Nested result sets

When a SELECT statement is sent to the database, a result set is returned. The sasql_fetch_row and sasql_fetch_array functions retrieve data from the individual rows of a result set, returning each row as an array of columns that can be queried further.

The source code for this sample is contained in your SQL Anywhere installation in a file called *nested.php*.


```
<?php
  # Connect using the default user ID and password
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      die ("Connection failed");
  } else {
      # Connected successfully.
  }
  # Retrieve the data and output HTML
  echo "<BR>\n";
  $query1 = "SELECT table_id, table_name FROM SYSTAB";
  $result = sasql_query( $conn, $query1 );
  if( $result ) {
    $num_rows = sasql_num_rows( $result );
    echo "Returned : $num_rows <BR>\n";
    $I = 1;
    while( ($row = sasql_fetch_array( $result ))) {
      echo "$I:  table_id:$row[table_id]" .
           " --- table_name:$row[table_name] <br>\n";
      $query2 = "SELECT table_id, column_name " .
      "FROM SYSTABCOL" .
      "WHERE table_id = '$row[table_id]'";
      echo " $query2 <br>\n";
      echo "  Columns: ";
      $result2 = sasql_query( $conn, $query2 );
      if( $result2 ) {
     while(($detailed = sasql_fetch_array($result2))) {
         echo " $detailed[column_name]";
      }
      sasql_free_result( $result2 );
       } else {
      echo "******FAILED********";
       }
       echo "<br>\n";
       $I++;
    }
  }
  echo "<BR>\n";
  sasql_disconnect( $conn );
?>
```

In the above sample, the SQL statement selects the table ID and name for each table from SYSTAB. The sasql_query function returns an array of rows. The script iterates through the rows using the sasql_fetch_array function to retrieve the rows from an array. An inner iteration goes through the columns of each row and prints their values.

Web forms

PHP can take user input from a web form, pass it to the database server as a SQL query, and display the result that is returned. The following example demonstrates a simple web form that gives the user the ability to query the sample database using SQL statements and display the results in an HTML table.

The source code for this sample is contained in your SQL Anywhere installation in a file called *webisql.php*.

```
<?php
  echo "<HTML>\n";
  $qname = $_POST["qname"];
  $qname = str_replace( "\\", "", $qname );
```

```
  echo "<form method=post action=webisql.php>\n";
  echo "<br>Query: <input type=text Size=80 name=qname value=\"$qname\">\n";
  echo "<input type=submit>\n";
  echo "</form>\n";
  echo "<HR><br>\n";
  if( ! $qname ) {
      echo "No Current Query\n";
      return;
  }
  # Connect to the database
  $con_str = "UID=DBA;PWD=sql;ENG=demo;LINKS=tcpip";
  $conn = sasql_connect( $con_str );
  if( ! $conn ) {
      echo "sasql_connect failed\n";
      echo "</html>\n";
      return 0;
  }
  $qname = str_replace( "\\", "", $qname );
  $result = sasql_query( $conn, $qname );
  if( ! $result ) {
        echo "sasql_query failed!";
  } else {
        // echo "query completed successfully\n";
        sasql_result_all( $result, "border=1" );
      sasql_free_result( $result );
  }
  sasql_disconnect( $conn );
  echo "</html>\n";
?>
```

This design could be extended to handle complex web forms by formulating customized SQL queries based on the values entered by the user.

Working with BLOBs

SQL Anywhere databases can store any type of data as a binary large object (BLOB). If that data is of a type readable by a web browser, a PHP script can easily retrieve it from the database and display it on a dynamically generated page.

BLOB fields are often used for storing non-text data, such as images in GIF or JPG format. Numerous types of data can be passed to a web browser without any need for third-party software or data type conversion. The following sample illustrates the process of adding an image to the database and then retrieving it again to be displayed in a web browser.

This sample is similar to the sample code in the files *image_insert.php* and *image_retrieve.php* of your SQL Anywhere installation. These samples also illustrate the use of a BLOB column for storing images.

```
<?php
  $conn = sasql_connect( "UID=DBA;PWD=sql" )
        or die("Can not connect to database");
  $create_table = "CREATE TABLE images (ID INTEGER PRIMARY KEY, img IMAGE)";
  sasql_query( $conn, $create_table);
  $insert = "INSERT INTO images VALUES (99,
xp_read_file('ianywhere_logo.gif'))";
  sasql_query( $conn, $insert );
  $query = "SELECT img FROM images WHERE ID = 99";
  $result = sasql_query($conn, $query);
  $data = sasql_fetch_row($result);
```



```
    $img = $data[0];
    header("Content-type: image/gif");
    echo $img;
    sasql_disconnect($conn);
  ?>
```

To be able to send the binary data from the database directly to a web browser, the script must set the data's MIME type using the header function. In this case, the browser is told to expect a GIF image so it can display it correctly.

SQL Anywhere PHP API reference

The PHP API supports the following functions:

Connections

Queries

Result sets

Transactions

Statements

Miscellaneous

sasql_affected_rows

Prototype

int **sasql_affected_rows(** sasql_conn *$conn* **)**

Description

Returns the number of rows affected by the last SQL statement. This function is typically used for INSERT, UPDATE, or DELETE statements. For SELECT statements, use the sasql_num_rows function.

Parameters

$conn The connection resource returned by a connect function.

Returns

The number of rows affected.

Related functions

sasql_commit

Prototype

bool **sasql_commit(** sasql_conn *$conn* **)**

Description

Ends a transaction on the SQL Anywhere database and makes any changes made during the transaction permanent. Useful only when the auto_commit option is Off.

Parameters

$conn The connection resource returned by a connect function.

Returns

TRUE on success or FALSE on failure.

Related functions

- "sasql_rollback" on page 654
- "sasql_set_option" on page 655
- "sasql_pconnect" on page 651
- "sasql_disconnect" on page 643

sasql_close

Prototype

bool **sasql_close(** sasql_conn *$conn* **)**

Description

Closes a previously opened database connection.

Parameters

$conn The connection resource returned by a connect function.

Returns

TRUE on success or FALSE on failure.

sasql_connect

Prototype

sasql_conn **sasql_connect(** string *$con_str* **)**

Description

Establishes a connection to a SQL Anywhere database.

Parameters

$con_str A connection string as recognized by SQL Anywhere.

Returns

A positive SQL Anywhere connection resource on success, or an error and 0 on failure.

Related functions

- "sasql_pconnect" on page 651
- "sasql_disconnect" on page 643

sasql_data_seek

Prototype

bool **sasql_data_seek(** sasql_result *$result***,** int *row_num* **)**

Description

Positions the cursor on row *row_num* on the *$result* that was opened using sasql_query.

Parameters

$result The result resource returned by the sasql_query function.

row_num An integer that represents the new position of the cursor within the result resource. For example, specify 0 to move the cursor to the first row of the result set or 5 to move it to the sixth row. Negative numbers represent rows relative to the end of the result set. For example, -1 moves the cursor to the last row in the result set and -2 moves it to the second-last row.

Returns

TRUE on success or FALSE on error.

Related functions

- "sasql_fetch_field" on page 646
- "sasql_fetch_array" on page 645
- "sasql_fetch_assoc" on page 646
- "sasql_fetch_row" on page 648
- "sasql_fetch_object" on page 647
- "sasql_query" on page 652

sasql_disconnect

Prototype

bool **sasql_disconnect(** sasql_conn *$conn* **)**

Description

Closes a connection that has already been opened with sasql_connect.

Parameters

$conn The connection resource returned by a connect function.

Returns

TRUE on success or FALSE on error.

Related functions

- "sasql_connect" on page 642
- "sasql_pconnect" on page 651

sasql_error

Prototype

bool **sasql_error(** [sasql_conn *$conn*] **)**

Description

Returns the error text of the most recently executed SQL Anywhere PHP function. Error messages are stored per connection. If no *$conn* is specified, then sasql_error returns the last error message where no connection was available. For example, if you call sasql_connect and the connection fails, then call sasql_error with no parameter for *$conn* to get the error message. If you want to obtain the corresponding SQL Anywhere error code value, use the sasql_errorcode function.

Parameters

$conn The connection resource returned by a connect function.

Returns

A string describing the error.

Related functions

- "sasql_errorcode" on page 644
- "sasql_set_option" on page 655

sasql_errorcode

Prototype

bool **sasql_errorcode(** [sasql_conn *$conn*] **)**

Description

Returns the error code of the most-recently executed SQL Anywhere PHP function. Error codes are stored per connection. If no *$conn* is specified, then sasql_errorcode returns the last error code where no connection was available. For example, if you are calling sasql_connect and the connection fails, then call sasql_errorcode with no parameter for the *$conn* to get the error code. If you want to get the corresponding error message use the sasql_error function.

Parameters

$conn The connection resource returned by a connect function.

Returns

An integer representing a SQL Anywhere error code. An error code of 0 means success. A positive error code indicates success with warnings. A negative error code indicates failure.

Related functions

- "sasql_connect" on page 642
- "sasql_pconnect" on page 651
- "sasql_error" on page 644
- "sasql_set_option" on page 655

sasql_escape_string

Prototype

string **sasql_escape_string(** sasql_conn *$conn***,** string *$str* **)**

Description

Escapes all special characters in the supplied string. The special characters that are escaped are \r, \n, ', ", ;, \, and the NULL character.

Parameters

$conn The connection resource returned by a connect function.

$string The string to be escaped.

Returns

The escaped string.

Related functions

- "sasql_connect" on page 642

sasql_fetch_array

Prototype

array **sasql_fetch_array(** sasql_result *$result* [**,** int *$result_type*]**)**

Description

Fetches one row from the result set. This row is returned as an array that can be indexed by the column names or by the column indexes.

Parameters

$result The result resource returned by the sasql_query function.

$result_type This optional parameter is a constant indicating what type of array should be produced from the current row data. The possible values for this parameter are the constants SASQL_ASSOC, SASQL_NUM, or SASQL_BOTH. It defaults to SASQL_BOTH.

By using the SASQL_ASSOC constant this function will behave identically to the sasql_fetch_assoc function, while SASQL_NUM will behave identically to the sasql_fetch_row function. The final option SASQL_BOTH will create a single array with the attributes of both.

Returns

An array that represents a row from the result set, or FALSE when no rows are available.

Related functions

- "sasql_data_seek" on page 643
- "sasql_fetch_assoc" on page 646
- "sasql_fetch_field" on page 646
- "sasql_fetch_row" on page 648
- "sasql_fetch_object" on page 647

sasql_fetch_assoc

Prototype

array **sasql_fetch_assoc(** sasql_result *$result* **)**

Description

Fetches one row from the result set as an associative array.

Parameters

$result The result resource returned by the sasql_query function.

Returns

An associative array of strings representing the fetched row in the result set, where each key in the array represents the name of one of the result set's columns or FALSE if there are no more rows in resultset.

Related functions

- "sasql_data_seek" on page 643
- "sasql_fetch_field" on page 646
- "sasql_fetch_field" on page 646
- "sasql_fetch_row" on page 648
- "sasql_fetch_object" on page 647

sasql_fetch_field

Prototype

object **sasql_fetch_field(** sasql_result *$result* [**,** int *$field_offset*] **)**

Description

Returns an object that contains information about a specific column.

Parameters

$result The result resource returned by the sasql_query function.

$field_offset An integer representing the column/field on which you want to retrieve information. Columns are zero based; to get the first column, specify the value 0. If this parameter is omitted, then the next field object is returned.

Returns

An object that has the following properties:

- **ID** contains the field's number
- **name** contains the field's name
- **numeric** indicates whether or not the field is a numeric value
- **length** returns the field's native storage size.
- **type** returns the field's type
- **native_type** returns the field's native type. These are values like DT_FIXCHAR, DT_DECIMAL or DT_DATE. See "Embedded SQL data types" on page 518.
- **precision** returns the field's numeric precision. This property is only set for fields with native_type equal to DT_DECIMAL.
- **scale** returns the field's numeric scale. This property is only set for fields with native_type equal to DT_DECIMAL.

Related functions

- "sasql_data_seek" on page 643
- "sasql_fetch_array" on page 645
- "sasql_fetch_assoc" on page 646
- "sasql_fetch_row" on page 648
- "sasql_fetch_object" on page 647

sasql_fetch_object

Prototype

array **sasql_fetch_object(** sasql_result *$result* **)**

Description

Fetches one row from the result set. This row is returned as an array that can be indexed by the column name only.

Parameters

$result The result resource returned by the sasql_query function.

Returns

An array that represents a row from the result set, or FALSE when no rows are available.

Related functions

- "sasql_data_seek" on page 643
- "sasql_fetch_field" on page 646
- "sasql_fetch_array" on page 645
- "sasql_fetch_assoc" on page 646
- "sasql_fetch_row" on page 648

sasql_fetch_row

Prototype

array **sasql_fetch_row(** sasql_result *$result* **)**

Description

Fetches one row from the result set. This row is returned as an array that can be indexed by the column indexes only.

Parameters

$result The result resource returned by the sasql_query function.

Returns

An array that represents a row from the result set, or FALSE when no rows are available.

Related functions

- "sasql_data_seek" on page 643
- "sasql_fetch_field" on page 646
- "sasql_fetch_array" on page 645
- "sasql_fetch_assoc" on page 646
- "sasql_fetch_object" on page 647

sasql_field_count

Prototype

int **sasql_field_count(** sasql_conn *$conn* **)**

Description

Returns the number of columns (fields) the last result contains.

Parameters

$conn The connection resource returned by a connect function.

Returns

A positive number of columns, or FALSE if *$conn* is not valid.

sasql_free_result

Prototype

bool **sasql_free_result(** sasql_result *$result* **)**

Description

Frees database resources associated with a result resource returned from sasql_query.

Parameters

$result The result resource returned by the sasql_query function.

Returns

TRUE on success or FALSE on error.

Related functions

- "sasql_query" on page 652

sasql_get_client_info

Prototype

string **sasql_get_client_info( )**

Description

Returns the version information of the client.

Parameters

None

Returns

A string that represents the SQL Anywhere client software version. The returned string is of the form X.Y.Z.W where X is the major version number, Y is the minor version number, Z is the patch number, and W is the build number (for example, 10.0.1.3616).

sasql_insert_id

Prototype

int **sasql_insert_id(** sasql_conn *$conn* **)**

Description

Returns the last value inserted into an IDENTITY column or a DEFAULT AUTOINCREMENT column, or zero if the most recent insert was into a table that did not contain an IDENTITY or DEFAULT AUTOINCREMENT column.

The sasql_insert_id function is provided for compatibility with MySQL databases.

Parameters

$conn The connection resource returned by a connect function.

Returns

The ID generated for an AUTOINCREMENT column by a previous INSERT statement or zero if last insert did not affect an AUTOINCREMENT column. The function can return FALSE if the *$conn* is not valid.

sasql_message

Prototype

bool **sasql_message(** sasql_conn *$conn***,** string *$message* **)**

Description

Writes a message to the server console.

Parameters

$conn The connection resource returned by a connect function.

$message A message to be written to the server console.

Returns

TRUE on success or FALSE on failure.

sasql_next_result

Prototype

bool **sasql_next_result(** sasql_conn *$conn* **)**

Description

Prepares the next result set from the last query that executed on *$conn*.

Parameters

$conn The connection resource returned by a connect function.

Returns

FALSE if there is no other result set to be retrieved. TRUE if there is another result to be retrieved. Call sasql_use_result or sasql_store_result to retrieve the next result set.

Related functions

- "sasql_use_result" on page 664
- "sasql_store_result" on page 664

sasql_num_rows

Prototype

int **sasql_num_rows(** sasql_result *$result* **)**

Description

Returns the number of rows that the *$result* contains.

Parameters

$result The result resource returned by the sasql_query function.

Returns

A positive number if the number of rows is exact, or a negative number if it is an estimate. To get the exact number of rows, the database option row_counts must be set permanently on the database, or temporarily on the connection. See "sasql_set_option" on page 655.

Related functions

- "sasql_query" on page 652

sasql_pconnect

Prototype

sasql_conn **sasql_pconnect(** string *$con_str* **)**

Description

Establishes a persistent connection to a SQL Anywhere database. Because of the way Apache creates child processes, you may observe a performance gain when using sasql_pconnect instead of sasql_connect. Persistent connections may provide improved performance in a similar fashion to connection pooling. If your database server has a limited number of connections (for example, the personal database server is limited to 10 concurrent connections), caution should be exercised when using persistent connections. Persistent connections could be attached to each of the child processes, and if you have more child processes in Apache than there are available connections, you will receive connection errors.

Parameters

$con_str A connection string as recognized by SQL Anywhere.

Returns

A positive SQL Anywhere persistent connection resource on success, or an error and 0 on failure.

Related functions

- "sasql_connect" on page 642
- "sasql_disconnect" on page 643

sasql_prepare

Prototype

sasql_stmt **sasql_prepare(** sasql_conn *$conn*, string *$sql_str* **)**

Description

Prepares the supplied SQL string.

Parameters

$conn The connection resource returned by a connect function.

$sql_str The SQL string to be prepared. The string can include parameter markers by embedding question marks at the appropriate positions.

Returns

A statement object or FALSE on failure.

Related functions

- "sasql_connect" on page 642
- "sasql_pconnect" on page 651

sasql_query

Prototype

mixed **sasql_query(** sasql_conn *$conn*, string *$sql_str* [, int *$result_mode*] **)**

Description

Prepares and executes the SQL query *$sql_str* on the connection identified by *$conn* that has already been opened using sasql_connect or sasql_pconnect.

Parameters

$conn The connection resource returned by a connect function.

$sql_str A SQL statement supported by SQL Anywhere.

$result_mode Either SASQL_USE_RESULT, or SASQL_STORE_RESULT (the default).

Returns

FALSE on failure; TRUE on success for INSERT, UPDATE, DELETE, CREATE; sasql_result for SELECT.

Related functions

- "sasql_free_result" on page 649
- "sasql_fetch_array" on page 645
- "sasql_fetch_field" on page 646
- "sasql_fetch_object" on page 647
- "sasql_fetch_row" on page 648

sasql_real_query

Prototype

bool **sasql_real_query(** sasql_conn *$conn*, string *$query* **)**

Description

Executes a query against the database using the supplied connection resource. The query result can be retrieved or stored using sasql_store_result or sasql_use_result. sasql_field_count can be used to check if the query returns a result set or not.

Parameters

$conn The connection resource returned by a connect function.

$query An SQL query.

Returns

TRUE on success or FALSE on failure.

Related functions

- "sasql_store_result" on page 664
- "sasql_use_result" on page 664
- "sasql_field_count" on page 648

sasql_result_all

Prototype

bool **sasql_result_all(** resource *$result*
[, *$html_table_format_string*
[, *$html_table_header_format_string*
[, *$html_table_row_format_string*
[, *$html_table_cell_format_string*
]]]] **)**

Description

Fetches all results of the *$result* and generates an HTML output table with an optional formatting string.

Parameters

$result The result resource returned by the sasql_query function.

$html_table_format_string A format string that applies to HTML tables. For example, **"Border=1; Cellpadding=5"**. The special value none does not create an HTML table. This is useful if you want to customize your column names or scripts. If you do not want to specify an explicit value for this parameter, use NULL for the parameter value.

$html_table_header_format_string A format string that applies to column headings for HTML tables. For example, **"bgcolor=#FF9533"**. The special value none does not create an HTML table. This is useful if you want to customize your column names or scripts. If you do not want to specify an explicit value for this parameter, use NULL for the parameter value.

$html_table_row_format_string A format string that applies to rows within HTML tables. For example, **"onclick='alert('this')'"**. If you would like different formats that alternate, use the special token ><. The left side of the token indicates which format to use on odd rows and the right side of the token is used to format even rows. If you do not place this token in your format string, all rows have the same format. If you do not want to specify an explicit value for this parameter, use NULL for the parameter value.

$html_table_cell_format_string A format string that applies to cells within HTML table rows. For example, **"onclick='alert('this')'"**. If you do not want to specify an explicit value for this parameter, use NULL for the parameter value.

Returns

TRUE on success or FALSE on failure.

Related functions

- "sasql_query" on page 652

sasql_rollback

Prototype

bool **sasql_rollback(** sasql_conn *$conn* **)**

Description

Ends a transaction on the SQL Anywhere database and discards any changes made during the transaction. This function is only useful when the auto_commit option is Off.

Parameters

$conn The connection resource returned by a connect function.

Returns

TRUE on success or FALSE on failure.

Related functions

- “sasql_commit” on page 641
- “sasql_set_option” on page 655

sasql_set_option

Prototype

bool **sasql_set_option(** sasql_conn *$conn*, string *$option*, mixed *$value* **)**

Description

Sets the value of the specified option on the specified connection. You can set the value for the following options:

Name	Description	Default
auto_commit	When this option is set to on, the database server commits after executing each statement.	on
row_counts	When this option is set to FALSE, the sasql_num_rows function returns an estimate of the number of rows affected. If you want to obtain an exact count, set this option to TRUE.	FALSE
verbose_errors	When this option is set to TRUE, the PHP driver returns verbose errors. When this option is set to FALSE, you must call the sasql_error or sasql_errorcode functions to get further error information.	TRUE

You can change the default value for an option by including the following line in the *php.ini* file. In this example, the default value is set for the auto_commit option.

```
sqlanywhere.auto_commit=0
```

Parameters

$conn The connection resource returned by a connect function.

$option The name of the option you want to set.

$value The new option value.

Returns

TRUE on success or FALSE on failure.

Related functions

- “sasql_commit” on page 641
- “sasql_error” on page 644
- “sasql_errorcode” on page 644
- “sasql_num_rows” on page 651
- “sasql_rollback” on page 654

sasql_stmt_affected_rows

Prototype

int **sasql_stmt_affected_rows(** sasql_stmt *$stmt* **)**

Description

Returns the number of rows affected by executing the statement.

Parameters

$stmt A statement resource that was executed by sasql_stmt_execute.

Returns

The number of rows affected or FALSE on failure.

Related functions

- "sasql_stmt_execute" on page 658

sasql_stmt_bind_param

Prototype

bool **sasql_stmt_bind_param(** sasql_stmt *$stmt***,** string *$types***,** mixed **&***$var1* **[,** mixed **&***$var2* ..] **)**

Description

Binds PHP variables to statement parameters.

Parameters

$stmt A prepared statement resource that was returned by the sasql_prepare function.

$types A string that contains one or more characters specifying the types of the corresponding bind

$var The variable references.

Returns

TRUE if binding the variables was successful or FALSE otherwise.

Related functions

- "sasql_connect" on page 642

sasql_stmt_bind_param_ex

Prototype

bool **sasql_stmt_bind_param_ex(** sasql_stmt *$stmt***,** int *$param_number***,** mixed **&***$var***,** string *$type* **[,** bool *$is_null* **[,** int *$direction*]] **)**

Description

Binds a PHP variable to a statement parameter.

Parameters

$stmt A prepared statement resource that was returned by the sasql_prepare function.

$param_number The parameter number. This should be a number between 1 and sasql_stmt_param_count.

$var A PHP variable. Only references to PHP variables are allowed.

$type Type of the variable. This can be one of: **s** for string, **i** for integer, **d** for double, **b** for blobs.

$is_null Whether the value of the variable is NULL or not.

$direction Can be SASQL_D_INPUT, SASQL_D_OUTPUT, or SASQL_INPUT_OUTPUT.

Returns

TRUE if binding the variable was successful or FALSE otherwise.

Related functions

- "sasql_stmt_param_count" on page 662

sasql_stmt_bind_result

Prototype

bool **sasql_stmt_bind_result(** sasql_stmt *$stmt***,** mixed **&***$var1* **[,** mixed **&***$var2* ..] **)**

Description

Binds one or more PHP variables to result columns of a statement that was executed, and returns a result set.

Parameters

$stmt A statement resource that was executed by sasql_stmt_execute.

$var1 References to PHP variables that will be bound to result set columns returned by the sasql_stmt_fetch.

Returns

TRUE on success or FALSE on failure.

Related functions

- "sasql_stmt_execute" on page 658
- "sasql_stmt_fetch" on page 659

sasql_stmt_close

Prototype

bool **sasql_stmt_close(** sasql_stmt *$stmt* **)**

Description

Closes the supplied statement resource and frees any resources associated with it. This function will also free any result objects that were returned by the sasql_stmt_result_metadata.

Parameters

$stmt A prepared statement resource that was returned by the sasql_prepare function.

Returns

TRUE for success or FALSE on failure.

Related functions

- "sasql_stmt_result_metadata" on page 662
- "sasql_prepare" on page 652

sasql_stmt_data_seek

Prototype

bool **sasql_stmt_data_seek(** sasql_stmt *$stmt***,** int *$offset* **)**

Description

This function seeks to the specified offset in the result set.

Parameters

$stmt A statement resource.

$offset The offset in the result set. This is a number between 0 and sasql_stmt_num_rows minus 1.

Returns

TRUE on success or FALSE failure.

Related functions

- "sasql_stmt_num_rows" on page 661

sasql_stmt_execute

Prototype

bool **sasql_stmt_execute(** sasql_stmt *$stmt* **)**

Description

Executes the prepared statement. The sasql_stmt_result_metadata can be used to check whether the statement returns a result set.

Parameters

$stmt A prepared statement resource that was returned by the sasql_prepare function. Variables should be bound before calling execute.

Returns

TRUE for success or FALSE on failure.

Related functions

- "sasql_prepare" on page 652
- "sasql_stmt_result_metadata" on page 662

sasql_stmt_fetch

Prototype

bool **sasql_stmt_fetch(** sasql_stmt *$stmt* **)**

Description

This function fetches one row out of the result for the statement and places the columns in the variables that were bound using sasql_stmt_bind_result.

Parameters

$stmt A statement resource.

Returns

TRUE on success or FALSE on failure.

Related functions

- "sasql_stmt_bind_result" on page 657

sasql_stmt_field_count

Prototype

int **sasql_stmt_field_count(** sasql_stmt *$stmt* **)**

Description

This function returns the number of columns in the result set of the statement.

Parameters

$stmt A statement resource.

Returns

The number of columns in the result of the statement. If the statement does not return a result, it returns 0.

Related functions

- "sasql_stmt_result_metadata" on page 662

sasql_stmt_free_result

Prototype

bool **sasql_stmt_free_result(** sasql_stmt *$stmt* **)**

Description

This function frees cached result set of the statement.

Parameters

$stmt A statement resource that was executed using sasql_stmt_execute.

Returns

TRUE on success or FALSE on failure.

Related functions

- "sasql_stmt_execute" on page 658
- "sasql_stmt_store_result" on page 663

sasql_stmt_insert_id

Prototype

int **sasql_stmt_insert_id(** sasql_stmt *$stmt* **)**

Description

Returns the last value inserted into an IDENTITY column or a DEFAULT AUTOINCREMENT column, or zero if the most recent insert was into a table that did not contain an IDENTITY or DEFAULT AUTOINCREMENT column.

Parameters

$stmt A statement resource that was executed by sasql_stmt_execute.

Returns

The ID generated for an IDENTITY column or a DEFAULT AUTOINCREMENT column by a previous INSERT statement, or zero if the last insert did not affect an IDENTITY or DEFAULT AUTOINCREMENT column. The function can return FALSE (0) if $stmt is not valid.

Related functions

- "sasql_stmt_execute" on page 658

sasql_stmt_next_result

Prototype

bool **sasql_stmt_next_result(** sasql_stmt *$stmt* **)**

Description

This function advances to the next result from the statement. If there is another result set, the currently cashed results are discarded and the associated result set object deleted (as returned by sasql_stmt_result_metadata).

Parameters

$stmt A statement resource.

Returns

TRUE on success or FALSE failure.

Related functions

- "sasql_stmt_result_metadata" on page 662

sasql_stmt_num_rows

Prototype

int **sasql_stmt_num_rows(** sasql_stmt *$stmt* **)**

Description

Returns the number of rows in the result set. The actual number of rows in the result set can only be determined after the sasql_stmt_store_result function is called to buffer the entire result set. If the sasql_stmt_store_result function has not been called, 0 is returned.

Parameters

$stmt A statement resource that was executed by sasql_stmt_execute and sasql_stmt_store_result was called on.

Returns

The number of rows available in the result or 0 on failure.

Related functions

- "sasql_stmt_execute" on page 658
- "sasql_stmt_store_result" on page 663

sasql_stmt_param_count

Prototype

int **sasql_stmt_param_count(** sasql_stmt *$stmt* **)**

Description

Returns the number of parameters in the supplied prepared statement handle

Parameters

$stmt A statement resource returns by the sasql_prepare function.

Returns

The number of parameters or FALSE on error.

Related functions

- "sasql_prepare" on page 652

sasql_stmt_reset

Prototype

bool **sasql_stmt_reset(** sasql_stmt *$stmt* **)**

Description

This function resets the *$stmt* object to the state just after the describe. Any variables that were bound are unbound and any data sent using sasql_stmt_send_long_data are dropped.

Parameters

$stmt A statement resource.

Returns

TRUE on success or FALSE on failure.

Related functions

- "sasql_stmt_send_long_data" on page 663

sasql_stmt_result_metadata

Prototype

sasql_result **sasql_stmt_result_metadata(** sasql_stmt *$stmt* **)**

Description

Returns a result set object for the supplied statement.

Parameters

$stmt A statement resource that was prepared and executed.

Returns

sasql_result object or FALSE if the statement does not return any results.

sasql_stmt_send_long_data

Prototype

bool **sasql_stmt_send_long_data(** sasql_stmt *$stmt*, int *$param_number*, string *$data* **)**

Description

Allows the user to send parameter data in chunks. The user must first call sasql_stmt_bind_param or sasql_stmt_bind_param_ex before attempting to send any data. The bind parameter must be of type string or blob. Repeatedly calling this function appends on to what was previously sent.

Parameters

$stmt A statement resource that was prepared using sasql_prepare.

$param_number The parameter number. This must be a number between 0 and (sasql_stmt_param_count() - 1).

$data The data to be sent.

Returns

TRUE on success or FALSE on failure.

Related functions

- "sasql_stmt_bind_param" on page 656
- "sasql_stmt_bind_param_ex" on page 656
- "sasql_prepare" on page 652
- "sasql_stmt_param_count" on page 662

sasql_stmt_store_result

Prototype

bool **sasql_stmt_store_result(** sasql_stmt *$stmt* **)**

Description

This function allows the client to cache the whole result set of the statement. You can use the function sasql_stmt_free_result to free the cached result.

Parameters

$stmt A statement resource that was executed using sasql_stmt_execute.

Returns

TRUE on success or FALSE on failure.

Related functions

- "sasql_stmt_free_result" on page 660
- "sasql_stmt_execute" on page 658

sasql_store_result

Prototype

sasql_result **sasql_store_result(** sasql_conn *$conn* **)**

Description

Transfers the result set from the last query on the database connection *$conn* to be used with the sasql_data_seek function.

Parameters

$conn The connection resource returned by a connect function.

Returns

FALSE if the query does not return a result object, or a result set object, that contains all the rows of the result. The result is cached at the client.

Related functions

- "sasql_data_seek" on page 643
- "sasql_stmt_execute" on page 658

sasql_use_result

Prototype

sasql_result **sasql_use_result(** sasql_conn *$conn* **)**

Description

Initiates a result set retrieval for the last query that executed on the connection.

Parameters

$conn The connection resource returned by a connect function.

Returns

FALSE if the query does not return a result object or a result set object. The result is not cached on the client.

Related functions

- "sasql_data_seek" on page 643

- “sasql_stmt_execute” on page 658

Deprecated PHP functions

The following PHP functions are supported but deprecated. Each of these functions has a newer equivalent with a name starting with sasql_ instead of sqlanywhere_.

sqlanywhere_commit (deprecated)

Prototype

bool **sqlanywhere_commit(** resource *link_identifier* **)**

Description

This function is deprecated. You should use the following PHP function instead:
“sasql_commit” on page 641.

Ends a transaction on the SQL Anywhere database and makes any changes made during the transaction permanent. Useful only when the auto_commit option is Off.

Parameters

link_identifier The link identifier returned by the sqlanywhere_connect function.

Returns

TRUE on success or FALSE on failure.

Example

This example shows how sqlanywhere_commit can be used to cause a commit on a specific connection.

```
$result = sqlanywhere_commit( $conn );
```

Related functions

- “sasql_commit” on page 641
- “sasql_pconnect” on page 651
- “sasql_disconnect” on page 643
- “sqlanywhere_pconnect (deprecated)” on page 676
- “sqlanywhere_disconnect (deprecated)” on page 667

sqlanywhere_connect (deprecated)

Prototype

resource **sqlanywhere_connect(** string *con_str* **)**

Description

This function is deprecated. You should use the following PHP function instead: "sasql_connect" on page 642.

Establishes a connection to a SQL Anywhere database.

Parameters

con_str A connection string as recognized by SQL Anywhere.

Returns

A positive SQL Anywhere link identifier on success, or an error and 0 on failure.

Example

This example passes the user ID and password for a SQL Anywhere database in the connection string.

```
$conn = sqlanywhere_connect( "UID=DBA;PWD=sql" );
```

Related functions

- "sasql_connect" on page 642
- "sasql_pconnect" on page 651
- "sasql_disconnect" on page 643
- "sqlanywhere_pconnect (deprecated)" on page 676
- "sqlanywhere_disconnect (deprecated)" on page 667

sqlanywhere_data_seek (deprecated)

Prototype

bool **sqlanywhere_data_seek(** resource *result_identifier*, int *row_num* **)**

Description

This function is deprecated. You should use the following PHP function instead: "sasql_data_seek" on page 643.

Positions the cursor on row *row_num* on the *result_identifier* that was opened using sqlanywhere_query.

Parameters

result_identifier The result resource returned by the sqlanywhere_query function.

row_num An integer that represents the new position of the cursor within the result_identifier. For example, specify 0 to move the cursor to the first row of the result set or 5 to move it to the sixth row. Negative numbers represent rows relative to the end of the result set. For example, -1 moves the cursor to the last row in the result set and -2 moves it to the second-last row.

Returns

TRUE on success or FALSE on error.

Example

This example shows how to seek to the sixth record in the result set.

```
sqlanywhere_data_seek( $result, 5 );
```

Related functions

- "sasql_data_seek" on page 643
- "sasql_fetch_field" on page 646
- "sasql_fetch_array" on page 645
- "sasql_fetch_row" on page 648
- "sasql_fetch_object" on page 647
- "sasql_query" on page 652
- "sqlanywhere_fetch_field (deprecated)" on page 671
- "sqlanywhere_fetch_array (deprecated)" on page 670
- "sqlanywhere_fetch_row (deprecated)" on page 673
- "sqlanywhere_fetch_object (deprecated)" on page 672
- "sqlanywhere_query (deprecated)" on page 677

sqlanywhere_disconnect (deprecated)

Prototype

bool **sqlanywhere_disconnect(** resource *link_identifier* **)**

Description

This function is deprecated. You should use the following PHP function instead: "sasql_disconnect" on page 643.

Closes a connection that has already been opened with sqlanywhere_connect.

Parameters

link_identifier The link identifier returned by the sqlanywhere_connect function.

Returns

TRUE on success or FALSE on error.

Example

This example closes the connection to a database.

```
sqlanywhere_disconnect( $conn );
```

Related functions

- "sasql_disconnect" on page 643
- "sasql_connect" on page 642
- "sasql_pconnect" on page 651
- "sqlanywhere_connect (deprecated)" on page 665
- "sqlanywhere_pconnect (deprecated)" on page 676

sqlanywhere_error (deprecated)

Prototype

string **sqlanywhere_error(** [resource *link_identifier*] **)**

Description

This function is deprecated. You should use the following PHP function instead:
"sasql_error" on page 644.

Returns the error text of the most recently executed SQL Anywhere PHP function. Error messages are stored per connection. If no *link_identifier* is specified, then sqlanywhere_error returns the last error message where no connection was available. For example, if you call sqlanywhere_connect and the connection fails, then call sqlanywhere_error with no parameter for *link_identifier* to get the error message. If you want to obtain the corresponding SQL Anywhere error code value, use the sqlanywhere_errorcode function.

Parameters

link_identifier A link identifier that was returned by sqlanywhere_connect or sqlanywhere_pconnect.

Returns

A string describing the error.

Example

This example attempts to select from a table that does not exist. The sqlanywhere_query function returns FALSE and the sqlanywhere_error function returns the error message.

```
$result = sqlanywhere_query( $conn, "SELECT * FROM
table_that_does_not_exist" );
if( ! $result ) {
      $error_msg = sqlanywhere_error( $conn );
      echo "Query failed. Reason: $error_msg";
}
```

Related functions

- "sasql_error" on page 644
- "sasql_errorcode" on page 644
- "sasql_set_option" on page 655
- "sqlanywhere_errorcode (deprecated)" on page 668
- "sqlanywhere_set_option (deprecated)" on page 680

sqlanywhere_errorcode (deprecated)

Prototype

bool **sqlanywhere_errorcode(** [resource *link_identifier*] **)**

Description

This function is deprecated. You should use the following PHP function instead:
"sasql_errorcode" on page 644.

Returns the error code of the most-recently executed SQL Anywhere PHP function. Error codes are stored per connection. If no *link_identifier* is specified, then sqlanywhere_errorcode returns the last error code where no connection was available. For example, if you are calling sqlanywhere_connect and the connection fails, then call sqlanywhere_errorcode with no parameter for the *link_identifier* to get the error code. If you want to get the corresponding error message use the sqlanywhere_error function.

Parameters

link_identifier A link identifier that was returned by sqlanywhere_connect or sqlanywhere_pconnect.

Returns

An integer representing a SQL Anywhere error code. An error code of 0 means success. A positive error code indicates success with warnings. A negative error code indicates failure.

Example

This example shows how you can retrieve the last error code from a failed SQL Anywhere PHP call.

```
$result = sqlanywhere_query( $conn, "SELECT * from
table_that_does_not_exist" );
   if( ! $result ) {
       $error_code = sqlanywhere_errorcode( $conn );
       echo "Query failed: Error code: $error_code";
   }
```

Related functions

- "sasql_error" on page 644
- "sasql_set_option" on page 655
- "sqlanywhere_error (deprecated)" on page 668
- "sqlanywhere_set_option (deprecated)" on page 680

sqlanywhere_execute (deprecated)

Prototype

bool **sqlanywhere_execute(** resource *link_identifier*, string *sql_str* **)**

Description

This function is deprecated.

Prepares and executes the SQL query *sql_str* on the connection identified by the *link_identifier* that has already been opened using sqlanywhere_connect or sqlanywhere_pconnect. This function returns TRUE or FALSE depending on the outcome of the query execution. This function is suitable for queries that do not return result sets. If you are expecting a result set, use the sqlanywhere_query function instead.

Parameters

link_identifier A link identifier returned by sqlanywhere_connect or sqlanywhere_pconnect.

sql_str A SQL query.

Returns

TRUE if the query executed successfully, otherwise, FALSE and an error message.

Example

This example shows how to execute a DDL statement using the sqlanywhere_execute function.

```
if( sqlanywhere_execute( $conn, "CREATE TABLE my_test_table( INT id )" ) ) {
   // handle success
} else {
   // handle failure
}
```

Related functions

- "sasql_query" on page 652
- "sqlanywhere_query (deprecated)" on page 677

sqlanywhere_fetch_array (deprecated)

Prototype

array **sqlanywhere_fetch_array(** resource *result_identifier* **)**

Description

This function is deprecated. You should use the following PHP function instead:
"sasql_fetch_array" on page 645.

Fetches one row from the result set. This row is returned as an array that can be indexed by the column names or by the column indexes.

Parameters

result_identifier The result resource returned by the sqlanywhere_query function.

Returns

An array that represents a row from the result set, or FALSE when no rows are available.

Example

This example shows how to retrieve all the rows in a result set. Each row is returned as an array.

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM
Employees" );
  While( ($row = sqlanywhere_fetch_array( $result )) ) {
 echo " GivenName = " . $row["GivenName"] . " \n" ;
       echo " Surname = $row[1] \n";
  }
```

Related functions

- "sasql_fetch_array" on page 645
- "sasql_data_seek" on page 643
- "sasql_fetch_field" on page 646

- "sasql_fetch_row" on page 648
- "sasql_fetch_object" on page 647
- "sasql_query" on page 652
- "sqlanywhere_data_seek (deprecated)" on page 666
- "sqlanywhere_fetch_field (deprecated)" on page 671
- "sqlanywhere_fetch_row (deprecated)" on page 673
- "sqlanywhere_fetch_object (deprecated)" on page 672
- "sqlanywhere_query (deprecated)" on page 677

sqlanywhere_fetch_field (deprecated)

Prototype

object **sqlanywhere_fetch_field(** resource *result_identifier* [, *field_offset*] **)**

Description

This function is deprecated. You should use the following PHP function instead: "sasql_fetch_field" on page 646.

Returns an object that contains information about a specific column.

Parameters

result_identifier The result resource returned by the sqlanywhere_query function.

field_offset An integer representing the column/field on which you want to retrieve information. Columns are zero based; to get the first column, specify the value 0. If this parameter is omitted, then the next field object is returned.

Returns

An object that has the following properties:

- **ID** contains the field/column number
- **name** contains the field/column name
- **numeric** indicates whether or not the field is a numeric value
- **length** returns field length
- **type** returns field type

Example

This example shows how to use sqlanywhere_fetch_field to retrieve all the column information for a result set.

```
$result = sqlanywhere_query($conn, "SELECT GivenName, Surname FROM
Employees");
while( ($field = sqlanywhere_fetch_field( $result )) ) {
        echo " Field ID = $field->id \n";
        echo " Field name = $field->name \n";
  }
```

Related functions

- "sasql_data_seek" on page 643
- "sasql_fetch_field" on page 646
- "sasql_fetch_array" on page 645
- "sasql_fetch_row" on page 648
- "sasql_fetch_object" on page 647
- "sasql_query" on page 652
- "sqlanywhere_data_seek (deprecated)" on page 666
- "sqlanywhere_fetch_array (deprecated)" on page 670
- "sqlanywhere_fetch_row (deprecated)" on page 673
- "sqlanywhere_fetch_object (deprecated)" on page 672
- "sqlanywhere_query (deprecated)" on page 677

sqlanywhere_fetch_object (deprecated)

Prototype

object **sqlanywhere_fetch_object(** resource *result_identifier* **)**

Description

This function is deprecated. You should use the following PHP function instead: "sasql_fetch_object" on page 647.

Fetches one row from the result set. This row is returned as an object that can be indexed by the column name only.

Parameters

result_identifier The result resource returned by the sqlanywhere_query function.

Returns

An object that represents a row from the result set, or FALSE when no rows are available.

Example

This example shows how to retrieve one row at a time from a result set as an object. Column names can be used as object members to access the column value.

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM
Employees" );
  While( ($row = sqlanywhere_fetch_object( $result )) ) {
        echo "$row->GivenName \n";   # output the data in the first column
only.
  }
```

Related functions

- "sasql_data_seek" on page 643
- "sasql_fetch_field" on page 646
- "sasql_fetch_array" on page 645
- "sasql_fetch_row" on page 648

- "sasql_fetch_object" on page 647
- "sasql_query" on page 652
- "sqlanywhere_query (deprecated)" on page 677
- "sqlanywhere_data_seek (deprecated)" on page 666
- "sqlanywhere_fetch_field (deprecated)" on page 671
- "sqlanywhere_fetch_array (deprecated)" on page 670
- "sqlanywhere_fetch_row (deprecated)" on page 673

sqlanywhere_fetch_row (deprecated)

Prototype

array **sqlanywhere_fetch_row(** resource *result_identifier* **)**

Description

This function is deprecated. You should use the following PHP function instead: "sasql_fetch_row" on page 648.

Fetches one row from the result set. This row is returned as an array that can be indexed by the column indexes only.

Parameters

result_identifier The result resource returned by the sqlanywhere_query function.

Returns

An array that represents a row from the result set, or FALSE when no rows are available.

Example

This example shows how to retrieve one row at a time from a result set.

```
while( ($row = sqlanywhere_fetch_row( $result )) ) {
        echo "$row[0] \n";     # output the data in the first column only.
  }
```

Related functions

- "sasql_fetch_row" on page 648
- "sasql_data_seek" on page 643
- "sasql_fetch_field" on page 646
- "sasql_fetch_array" on page 645
- "sasql_fetch_object" on page 647
- "sasql_query" on page 652
- "sqlanywhere_data_seek (deprecated)" on page 666
- "sqlanywhere_fetch_field (deprecated)" on page 671
- "sqlanywhere_fetch_array (deprecated)" on page 670
- "sqlanywhere_fetch_object (deprecated)" on page 672
- "sqlanywhere_query (deprecated)" on page 677

sqlanywhere_free_result (deprecated)

Prototype

bool **sqlanywhere_free_result(** resource *result_identifier* **)**

Description

This function is deprecated. You should use the following PHP function instead:
"sasql_free_result" on page 649.

Frees database resources associated with a result resource returned from sqlanywhere_query.

Parameters

result_identifier The result resource returned by the sqlanywhere_query function.

Returns

TRUE on success or FALSE on error.

Example

This example shows how to free a result identifier's resources.

```
sqlanywhere_free_result( $result );
```

Related functions

- "sasql_query" on page 652
- "sasql_free_result" on page 649
- "sqlanywhere_query (deprecated)" on page 677

sqlanywhere_identity (deprecated)

Prototype

int **sqlanywhere_identity(** resource *link_identifier* **)**

int **sqlanywhere_insert_id(** resource *link_identifier* **)**

Description

This function is deprecated. You should use the following PHP function instead:
"sasql_insert_id" on page 649.

Returns the last value inserted into an IDENTITY column or a DEFAULT AUTOINCREMENT column, or zero if the most recent insert was into a table that did not contain an IDENTITY or DEFAULT AUTOINCREMENT column.

The sqlanywhere_insert_id function is provided for compatibility with MySQL databases.

Parameters

link_identifier A link identifier returned by sqlanywhere_connect or sqlanywhere_pconnect.

Returns

The ID generated for an AUTOINCREMENT column by a previous INSERT statement or zero if last insert did not affect an AUTOINCREMENT column. The function can return FALSE if the *link_identifier* is not valid.

Example

This example shows how the sqlanywhere_identity function can be used to retrieve the autoincrement value most recently inserted into a table by the specified connection.

```
if( sqlanywhere_execute( $conn, "INSERT INTO my_auto_increment_table VALUES
( 1 ) " ) ) {
      $insert_id = sqlanywhere_insert_id( $conn );
      echo "Last insert id = $insert_id";
}
```

Related functions

- "sasql_insert_id" on page 649
- "sqlanywhere_execute (deprecated)" on page 669

sqlanywhere_num_fields (deprecated)

Prototype

int **sqlanywhere_num_fields(** resource *result_identifier* **)**

Description

This function is deprecated. You should use the following PHP function instead: "sasql_field_count" on page 648.

Returns the number of columns (fields) the *result_identifier* contains.

Parameters

result_identifier The result resource returned by the sqlanywhere_query function.

Returns

A positive number of columns, or an error if *result_identifier* is not valid.

Example

This example returns a value indicating how many columns are in the result set.

```
$num_columns = sqlanywhere_num_fields( $result );
```

Related functions

- "sasql_field_count" on page 648
- "sasql_query" on page 652
- "sqlanywhere_query (deprecated)" on page 677

sqlanywhere_num_rows (deprecated)

Prototype

int **sqlanywhere_num_rows(** resource *result_identifier* **)**

Description

This function is deprecated. You should use the following PHP function instead:
"sasql_num_rows" on page 651.

Returns the number of rows that the *result_identifier* contains.

Parameters

result_identifier The result resource returned by the sqlanywhere_query function.

Returns

A positive number if the number of rows is exact, or a negative number if it is an estimate. To get the exact number of rows, the database option row_counts must be set permanently on the database, or temporarily on the connection. See "sasql_set_option" on page 655.

Example

This example shows how to retrieve the estimated number of rows returned in a result set:

```
$num_rows = sqlanywhere_num_rows( $result );
   if( $num_rows < 0 ) {
       $num_rows = abs( $num_rows );    # take the absolute value as an
estimate
   }
```

Related functions

- "sasql_num_rows" on page 651
- "sasql_query" on page 652
- "sqlanywhere_query (deprecated)" on page 677

sqlanywhere_pconnect (deprecated)

Prototype

resource **sqlanywhere_pconnect(** string *con_str* **)**

Description

This function is deprecated. You should use the following PHP function instead:
"sasql_pconnect" on page 651.

Establishes a persistent connection to a SQL Anywhere database. Because of the way Apache creates child processes, you may observe a performance gain when using sqlanywhere_pconnect instead of sqlanywhere_connect. Persistent connections may provide improved performance in a similar fashion to connection pooling. If your database server has a limited number of connections (for example, the personal database server is limited to 10 concurrent connections), caution should be exercised when using persistent

connections. Persistent connections could be attached to each of the child processes, and if you have more child processes in Apache than there are available connections, you will receive connection errors.

Parameters

con_str A connection string as recognized by SQL Anywhere.

Returns

A positive SQL Anywhere persistent link identifier on success, or an error and 0 on failure.

Example

This example shows how to retrieve all the rows in a result set. Each row is returned as an array.

```
$conn = sqlanywhere_pconnect( "UID=DBA;PWD=sql" );
```

Related functions

- "sasql_pconnect" on page 651
- "sasql_connect" on page 642
- "sasql_disconnect" on page 643
- "sqlanywhere_connect (deprecated)" on page 665
- "sqlanywhere_disconnect (deprecated)" on page 667

sqlanywhere_query (deprecated)

Prototype

resource **sqlanywhere_query(** resource *link_identifier*, string *sql_str* **)**

Description

This function is deprecated. You should use the following PHP function instead: "sasql_query" on page 652.

Prepares and executes the SQL query *sql_str* on the connection identified by *link_identifier* that has already been opened using sqlanywhere_connect or sqlanywhere_pconnect. For queries that do not return result sets, you can use the sqlanywhere_execute function.

Parameters

link_identifier The link identifier returned by the sqlanywhere_connect function.

sql_str A SQL statement supported by SQL Anywhere.

For more information about SQL statements, see "SQL statements" [*SQL Anywhere Server - SQL Reference*].

Returns

A positive value representing the result resource on success, or 0 and an error message on failure.

Example

This example executes the query SELECT * FROM SYSTAB on the SQL Anywhere database.

```
$result = sqlanywhere_query( $conn, "SELECT * FROM SYSTAB" );
```

Related functions

- "sasql_query" on page 652
- "sasql_free_result" on page 649
- "sasql_fetch_array" on page 645
- "sasql_fetch_field" on page 646
- "sasql_fetch_object" on page 647
- "sasql_fetch_row" on page 648
- "sqlanywhere_execute (deprecated)" on page 669
- "sqlanywhere_free_result (deprecated)" on page 674
- "sqlanywhere_fetch_array (deprecated)" on page 670
- "sqlanywhere_fetch_field (deprecated)" on page 671
- "sqlanywhere_fetch_object (deprecated)" on page 672
- "sqlanywhere_fetch_row (deprecated)" on page 673

sqlanywhere_result_all (deprecated)

Prototype

bool **sqlanywhere_result_all(** resource *result_identifier* **[,** *html_table_format_string* **[,** *html_table_header_format_string* **[,** *html_table_row_format_string* **[,** *html_table_cell_format_string*]]]] **)**

Description

This function is deprecated.

Fetches all results of the *result_identifier* and generates an HTML output table with an optional formatting string.

Parameters

result_identifier The result resource returned by the sqlanywhere_query function.

html_table_format_string A format string that applies to HTML tables. For example, **"Border=1; Cellpadding=5"**. The special value none does not create an HTML table. This is useful if you want to customize your column names or scripts. If you do not want to specify an explicit value for this parameter, use NULL for the parameter value.

html_table_header_format_string A format string that applies to column headings for HTML tables. For example, **"bgcolor=#FF9533"**. The special value none does not create an HTML table. This is useful if you want to customize your column names or scripts. If you do not want to specify an explicit value for this parameter, use NULL for the parameter value.

html_table_row_format_string A format string that applies to rows within HTML tables. For example, **"onclick='alert('this')'"**. If you would like different formats that alternate, use the special token ><. The left side of the token indicates which format to use on odd rows and the right side of the token is used to format even rows. If you do not place this token in your format string, all rows have the same format. If you do not want to specify an explicit value for this parameter, use NULL for the parameter value.

html_table_cell_format_string A format string that applies to cells within HTML table rows. For example, **"onclick='alert('this')'"**. If you do not want to specify an explicit value for this parameter, use NULL for the parameter value.

Returns

TRUE on success or FALSE on failure.

Example

This example shows how to use sqlanywhere_result_all to generate an HTML table with all the rows from a result set.

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM
Employees" );
  sqlanywhere_result_all( $result );
```

This example shows how to use different formatting on alternate rows using a style sheet.

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM
Employees");
sqlanywhere_result_all( $result, "border=2", "bordercolor=#3F3986",
"bgcolor=#3F3986 style=\"color=#FF9533\"", 'class="even"><class="odd"');
```

Related functions

- "sasql_query" on page 652
- "sqlanywhere_query (deprecated)" on page 677

sqlanywhere_rollback (deprecated)

Prototype

bool **sqlanywhere_rollback(** resource *link_identifier* **)**

Description

This function is deprecated. You should use the following PHP function instead:
"sasql_rollback" on page 654.

Ends a transaction on the SQL Anywhere database and discards any changes made during the transaction. This function is only useful when the auto_commit option is Off.

Parameters

link_identifier The link identifier returned by the sqlanywhere_connect function.

Returns

TRUE on success or FALSE on failure.

Example

This example uses sqlanywhere_rollback to roll back a connection.

```
$result = sqlanywhere_rollback( $conn );
```

Related functions

- "sasql_rollback" on page 654
- "sasql_commit" on page 641
- "sasql_set_option" on page 655
- "sqlanywhere_commit (deprecated)" on page 665
- "sqlanywhere_set_option (deprecated)" on page 680

sqlanywhere_set_option (deprecated)

Prototype

bool **sqlanywhere_set_option(** resource *link_identifier*, string *option*, mixed *value* **)**

Description

This function is deprecated. You should use the following PHP function instead: "sasql_set_option" on page 655.

Sets the value of the specified option on the specified connection. You can set the value for the following options:

Name	Description	Default
auto_commit	When this option is set to on, the database server commits after executing each statement.	on
row_counts	When this option is set to FALSE, the sqlanywhere_num_rows function returns an estimate of the number of rows affected. If you want to obtain an exact count, set this option to TRUE.	FALSE
verbose_errors	When this option is set to TRUE, the PHP driver returns verbose errors. When this option is set to FALSE, you must call the sqlanywhere_error or sqlanywhere_errorcode functions to get further error information.	TRUE

You can change the default value for an option by including the following line in the *php.ini* file. In this example, the default value is set for the auto_commit option.

```
sqlanywhere.auto_commit=0
```

Parameters

link_identifier The link identifier returned by the sqlanywhere_connect function.

option The name of the option you want to set.

value The new option value.

Returns

TRUE on success or FALSE on failure.

Example

The following examples show the different ways you can set the value of the auto_commit option.

```
$result = sqlanywhere_set_option( $conn, "auto_commit", "Off" );

$result = sqlanywhere_set_option( $conn, "auto_commit", 0 );

$result = sqlanywhere_set_option( $conn, "auto_commit", False );
```

Related functions

- “sasql_set_option” on page 655
- “sasql_commit” on page 641
- “sasql_error” on page 644
- “sasql_errorcode” on page 644
- “sasql_num_rows” on page 651
- “sasql_rollback” on page 654
- “sqlanywhere_commit (deprecated)” on page 665
- “sqlanywhere_error (deprecated)” on page 668
- “sqlanywhere_errorcode (deprecated)” on page 668
- “sqlanywhere_num_rows (deprecated)” on page 676
- “sqlanywhere_rollback (deprecated)” on page 679

CHAPTER 19

SQL Anywhere external environment support

Contents

Overview of external environments

SQL Anywhere includes support for six external runtime environments. These include embedded SQL and ODBC applications written in C/C++, as well as applications written in Java, Perl, PHP, or languages such as C# and Visual Basic that are based on Microsoft's Common Language Runtime (CLR).

SQL Anywhere has had the ability to call compiled native functions written in C or C++ for some time. However, when these procedures are run by the database server, the dynamic link library or shared object has always been loaded by the database server and the calls out to the native functions have always been made by the database server. The risk here is that if the native function causes a fault, then the database server crashes. Running compiled native functions outside of the database server, in an external environment, eliminates these risks to the server.

The following is an overview of the external environment support in SQL Anywhere.

Catalog tables

A system catalog table stores the information needed to identify and launch each of the external environments. The definition for this table is:

```
SYS.SYSEXTERNENV (
  object_id            unsigned bigint not null,
  name                 varchar(128)    not null,
  scope                char(1)         not null,
  supports_result_sets char(1)         not null,
  location             long varchar    not null,
  options              long varchar    not null,
  user_id              unsigned int
)
```

- **object_id** A unique identifier that is generated by the database server.
- **name** The name column identifies the name of the external environment or language. It is one of **java**, **perl**, **php**, **clr**, **c_esql32**, **c_esql64**, **c_odbc32**, or **c_odbc64**.
- **scope** The scope column is either **C** for CONNECTION or **D** for DATABASE respectively. The scope column identifies if the external environment is launched as one-per-connection or one-per-database.

  For one-per-connection external environments (like PERL, PHP, C_ESQL32, C_ESQL64, C_ODBC32, and C_ODBC64), there is one instance of the external environment for each connection using the external environment. In the case of one-per-connection, the external environment terminates when the connection terminates.

  For one-per-database external environments (like JAVA and CLR), there is one instance of the external environment for each database using the external environment. In the case of one-per-database, the external environment terminates when the database is stopped.
- **supports_result_sets** The supports_result_sets column identifies those external environments that can return result sets. All of the external environments can return result sets except PERL and PHP.
- **location** The location column identifies the location on the database server computer where the executable/binary for the external environment can be found. It includes the executable/binary name.

This path can either be fully qualified or relative. If the path is relative, then the executable/binary must be in a location where the database server can find it.

- **options** The options column identifies any options required on the command line to launch the executable associated with the external environment. You should not modify this column.
- **user_id** The user_id column identifies a user ID in the database that has DBA authority. When the external environment is initially launched, it must make a connection back to the database to set things up for the external environment's usage. By default, this connection is made using the DBA user ID, but if the database administrator prefers to have the external environment use a different user ID with DBA authority, then the user_id column in the SYS.SYSEXTERNENV table would indicate that different user ID instead. In most cases, though, this column in SYS.SYSEXTERNENV is NULL and the database server, by default, uses the DBA user ID.

Another system catalog table stores the non-Java external environment objects. The table definition for this table is:

```
SYS.SYSEXTERNENVOBJECT (
  object_id    unsigned bigint not null,
  extenv_id    unsigned bigint not null,
  owner        unsigned int    not null,
  name         long varchar    not null,
  contents     long binary     not null,
  update_time  timestamp       not null
)
```

- **object_id** A unique identifier that is generated by the database server.
- **extenv_id** The extenv_id identifies the external environment type (as stored in SYS.SYSEXTERNENV).
- **owner** The owner column identifies the creator/owner of the external environment object.
- **name** The name column is the name of the external environment object as specified in the INSTALL EXTERNAL OBJECTstatement.
- **contents** The contents column contains the contents of the external environment object.
- **update_time** The update_time column represents the last time the object was modified (or installed).

Deprecated options

With the introduction of the SYS.SYSEXTERNENV table, some Java-specific options have now been deprecated. These deprecated options are:

```
java_location
java_main_userid
```

Applications that have been using these options to identify which specific Java VM to use or which user ID to use for installing classes and other Java-related administrative tasks should use the ALTER EXTERNAL ENVIRONMENT statement instead to set the location and user_id values in the SYS.SYSEXTERNENV table for Java.

SQL statements

The following SQL syntax allows you to set or modify values in the SYS.SYSEXTERNENV table.

```
ALTER EXTERNAL ENVIRONMENT environment-name
    [ USER user-name ]
    [ LOCATION location-string ]
```

- **environment-name** The environment name is an identifier representing the name of the environment in SYS.SYSEXTERNENV. It is one of PERL, PHP, JAVA, CLR, C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64.
- **user-name** The user name string identifies a user in the database who has DBA authority. When the external environment is initially launched, it must make a connection back to the database to set things up for the external environment's usage. By default, this connection is made using the DBA user ID, but if the database administrator prefers to have the external environment use a different user ID with DBA authority, then user-name would indicate the different user ID to be used. In most cases, this option need not be specified.
- **location-string** The location string identifies the location on the database server computer where the executable/binary for the external environment can be found. It includes the executable/binary name. This path can either be fully qualified or relative. If the path is relative, then the executable/binary must be in a location where the database server can find it.

Once an external environment is set up to be used on the database server, you can then install objects into the database and create stored procedures and functions that make use of these objects within the external environment. Installation, creation, and usage of these objects, stored procedures, and stored functions is very similar to the current method of installing Java classes and creating and using Java stored procedures and functions.

To add a comment for an external environment, you can execute:

```
COMMENT ON EXTERNAL ENVIRONMENT environment-name
  IS comment-string
```

To install an external environment object (for example, a Perl script) from a file or an expression into the database, you would need to execute an INSTALL EXTERNAL OBJECT statement similar to the following:

```
INSTALL EXTERNAL OBJECT object-name-string
  [ update-mode ]
  FROM { FILE file-path | VALUE expression }
  ENVIRONMENT environment-name
```

- **object-name-string** The object name string is the name by which the installed object is identified within the database.
- **update-mode** The update mode is either NEW or UPDATE. If the update mode is omitted, then NEW is assumed.
- **file-path** The file path is the location on the database server computer from where the object is installed.
- **environment-name** The environment name is one of JAVA, PERL, PHP, CLR, C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64.

To add a comment for an installed external environment object, you can execute:

```
COMMENT ON EXTERNAL ENVIRONMENT OBJECT object-name-string
  IS comment-string
```


To remove an installed external environment from the database, you would need to use a REMOVE statement that is similar to the existing REMOVE JAVA statement:

```
REMOVE EXTERNAL OBJECT object-name-string
```

- **object-name-string** The object name string is the same string that was specified in the corresponding INSTALL EXTERNAL OBJECT statement.

Once the external environment objects are installed in the database, they can be used within external stored procedure and function definitions (similar to the current mechanism for creating Java stored procedures and functions).

```
CREATE PROCEDURE procedure-name(...)
  EXTERNAL NAME '...'
  LANGUAGE environment-name

CREATE FUNCTION function-name(...)
  RETURNS ...
  EXTERNAL NAME '...'
  LANGUAGE environment-name
```

- **environment-name** The environment name is one of JAVA, PERL, PHP, CLR, C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64.

Once these stored procedures and functions are created, they can be used like any other stored procedure or function in the database. The database server, when encountering an external environment stored procedure or function, automatically launches the external environment (if it has not already been started), and sends over whatever information is needed to get the external environment to fetch the external environment object from the database and execute it. Any result sets or return values resulting from the execution are returned as needed.

If you want to start or stop an external environment on demand, you can use the START EXTERNAL ENVIRONMENT and STOP EXTERNAL ENVIRONMENT statements (similar to the current START JAVA and STOP JAVA statements):

```
START EXTERNAL ENVIRONMENT environment-name
STOP EXTERNAL ENVIRONMENT environment-name
```

- **environment-name** Environment name is one of JAVA, PERL, PHP, CLR, C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64.

For more information, see:

- "ALTER EXTERNAL ENVIRONMENT statement" [*SQL Anywhere Server - SQL Reference*]
- "INSTALL EXTERNAL OBJECT statement" [*SQL Anywhere Server - SQL Reference*]
- "REMOVE EXTERNAL OBJECT statement" [*SQL Anywhere Server - SQL Reference*]
- "COMMENT statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE FUNCTION statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*]
- "START EXTERNAL ENVIRONMENT statement" [*SQL Anywhere Server - SQL Reference*]
- "STOP EXTERNAL ENVIRONMENT statement" [*SQL Anywhere Server - SQL Reference*]

The PERL external environment

SQL Anywhere includes support for Perl stored procedures and functions. A Perl stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in Perl and the execution of the procedure or function takes place outside of the database server (that is, within a Perl executable instance). It should be noted that there is a separate instance of the Perl executable for each connection that uses Perl stored procedures and functions. This behavior is different from Java stored procedures and functions. In the case of Java, there is once instance of the Java VM for each database rather than one instance per connection. The other major difference between Perl and Java is that Perl stored procedures do not return result sets, whereas Java stored procedures can return result sets.

There are a few prerequisites to using Perl in the database support:

1. A copy of Perl must be installed on the database server computer and the Perl executable must be locatable by the SQL Anywhere database server.
2. The DBD::SQLAnywhere driver must be installed on the database server computer.
3. On Windows, Microsoft Visual Studio must also be installed. This is a prerequisite since it is necessary for installing the DBD::SQLAnywhere driver.

For more information about installing the DBD::SQLAnywhere driver, see "SQL Anywhere Perl DBD::SQLAnywhere API" on page 607.

In addition to the above prerequisites, the database administrator must also install the SQL Anywhere Perl External Environment module. To install the external environment module:

To install the external environment module (Windows)

- Run the following commands from the *SDK\PerlEnv* subdirectory of your SQL Anywhere installation:

```
perl Makefile.PL
nmake
nmake install
```

To install the external environment module (Unix)

- Run the following commands from the *sdk/perlenv* subdirectory of your SQL Anywhere installation:

```
perl Makefile.PL
make
make install
```

Once the Perl external environment module has been built and installed, the Perl in the database support can be used. Note that Perl in the database support is only available with SQL Anywhere version 11 or later databases. If a SQL Anywhere 10 database is loaded, then an error indicating that external environments are not supported is returned when you try to use the Perl in the database support.

To use Perl in the database, make sure that the database server is able to locate and start the Perl executable. Verify that this can be done by executing:

```
START EXTERNAL ENVIRONMENT PERL;
```

If the database server fails to start Perl, then the problem probably occurs because the database server is not able to locate the Perl executable. In this case, you should execute an ALTER EXTERNAL

ENVIRONMENT statement to explicitly set the location of the Perl executable. Make sure to include the executable file name.

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'perl-path';
```

For example:

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'c:\\Perl\\bin\\perl.exe';
```

Note that the START EXTERNAL ENVIRONMENT PERL statement is not necessary other than to verify that the database server can start Perl. In general, making a Perl stored procedure or function call starts Perl automatically.

Similarly, the STOP EXTERNAL ENVIRONMENT PERL statement is not necessary to stop an instance of Perl since the instance automatically goes away when the connection terminates. However, if your are completely done with Perl and you want to free up some resources, then the STOP EXTERNAL ENVIRONMENT PERL statement releases the Perl instance for your connection.

Once you have verified that the database server can start the Perl executable, the next thing to do is to install the necessary Perl code into the database. Do this by using the INSTALL statement. For example, you can execute the following statement to install a Perl script from a file into the database.

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM FILE 'perl-file'
  ENVIRONMENT PERL;
```

Perl code also can be built and installed from an expression, as follows:

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL;
```

Perl code also can be built and installed from a variable, as follows:

```
CREATE VARIABLE PerlVariable LONG VARCHAR;
SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE PerlVariable
  ENVIRONMENT PERL;
```

To remove Perl code from the database, use the REMOVE statement, as follows:

```
REMOVE EXTERNAL OBJECT 'perl-script'
```

To modify existing Perl code, you can use the UPDATE clause of the INSTALL EXTERNAL OBJECT statement, as follows:

```
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM FILE 'perl-file'
  ENVIRONMENT PERL

INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
```

```
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL

SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM VALUE PerlVariable
  ENVIRONMENT PERL
```

Once the Perl code is installed in the database, you can then create the necessary Perl stored procedures and functions. When creating Perl stored procedures and functions, the LANGUAGE is always PERL and the EXTERNAL NAME string contains the information needed to call the Perl subroutines and to return OUT parameters and return values. The following global variables are available to the Perl code on each call:

- **$sa_perl_return** This is used to set the return value for a function call.
- **$sa_perl_argN** where N is a positive integer [0 .. n]. This is used for passing the SQL arguments down to the Perl code. For example, `$sa_perl_arg0` refers to argument 0, `$sa_perl_arg1` refers to argument 1, and so on.
- **$sa_perl_default_connection** This is used for making server-side Perl calls.
- **$sa_output_handle** This is used for sending output from the Perl code to the database server messages window.

A Perl stored procedure can be created with any set of data types for input and output arguments, and for the return value. However, all non-binary datatypes are mapped to strings when making the Perl call while binary data is mapped to an array of numbers. A simple Perl example follows:

```
INSTALL EXTERNAL OBJECT 'SimplePerlExample'
  NEW
  FROM VALUE 'sub SimplePerlSub{
    return( ($_[0] * 1000) +
            ($_[1] * 100) +
            ($_[2] * 10) +
            $_[3] );
  }'
  ENVIRONMENT PERL;

CREATE FUNCTION SimplePerlDemo(
    IN thousands INT,
    IN hundreds INT,
    IN tens INT,
    IN ones INT)
  RETURNS INT
  EXTERNAL NAME '<file=SimplePerlExample>
    $sa_perl_return = SimplePerlSub(
      $sa_perl_arg0,
      $sa_perl_arg1,
      $sa_perl_arg2,
      $sa_perl_arg3)'
  LANGUAGE PERL;

// The number 1234 should appear
SELECT SimplePerlDemo(1,2,3,4);
```

The following Perl example takes a string and writes it to the database server messages window:

```
INSTALL EXTERNAL OBJECT 'PerlConsoleExample'
  NEW
```



```
  FROM VALUE 'sub WriteToServerConsole { print $sa_output_handle $_[0]; }'
  ENVIRONMENT PERL;

CREATE PROCEDURE PerlWriteToConsole( IN str LONG VARCHAR)
  EXTERNAL NAME '<file=PerlConsoleExample>
    WriteToServerConsole( $sa_perl_arg0 )'
  LANGUAGE PERL;

// 'Hello world' should appear in the database server messages window
CALL PerlWriteToConsole( 'Hello world' );
```

To use server-side Perl, the Perl code must use the $sa_perl_default_connection variable. The following example creates a table and then calls a Perl stored procedure to populate the table:

```
CREATE TABLE perlTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePerlExample'
  NEW
  FROM VALUE 'sub ServerSidePerlSub
    { $sa_perl_default_connection->do(
        "INSERT INTO perlTab SELECT table_id, table_name FROM SYS.SYSTAB" );
      $sa_perl_default_connection->do(
        "COMMIT" );
    }'
  ENVIRONMENT PERL;

CREATE PROCEDURE PerlPopulateTable()
  EXTERNAL NAME '<file=ServerSidePerlExample> ServerSidePerlSub()'
  LANGUAGE PERL;

CALL PerlPopulateTable();

// The following should return 2 identical rows
SELECT count(*) FROM perlTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

For additional information and examples on using the Perl in the database support, refer to the examples located in the *samples-dir\SQLAnywhere\ExternalEnvironments\Perl* directory.

The PHP external environment

SQL Anywhere includes support for PHP stored procedures and functions. A PHP stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in PHP and the execution of the procedure or function takes place outside of the database server (that is, within a PHP executable instance). There is a separate instance of the PHP executable for each connection that uses PHP stored procedures and functions. This behavior is quite different from Java stored procedures and functions. In the case of Java, there is once instance of the Java VM for each database rather than one instance per connection. The other major difference between PHP and Java is that PHP stored procedures do not return result sets, whereas Java stored procedures can return result sets. PHP only returns an object of type LONG VARCHAR, which is the output of the PHP script.

There are two prerequisites to using PHP in the database support:

1. A copy of PHP must be installed on the database server computer and the PHP executable must be locatable by the database server.
2. The SQL Anywhere PHP driver (shipped with SQL Anywhere) must be installed on the database server computer. See "Installing and configuring SQL Anywhere PHP" on page 627.

In addition to the above two prerequisites, the database administrator must also install the SQL Anywhere PHP External Environment module. Prebuilt modules for several versions of PHP are included with the SQL Anywhere distribution. To install prebuilt modules, copy the appropriate driver module to your PHP extensions directory (which can be found in *php.ini*). On Unix, you can also use a symbolic link.

To install the external environment module (Windows)

1. Locate the *php.ini* file for your PHP installation, and open it in a text editor. Locate the line that specifies the location of the **extension_dir** directory. If **extension_dir** is not set to any specific directory, it is a good idea to set it to point to an isolated directory for better system security.
2. Copy the desired PHP module from the SQL Anywhere installation directory to your PHP extensions directory. The following is a model to use:

```
copy install-dir\Bin32\php-5.2.6_sqlanywhere_extenv11.dll
  php-dir\ext
```

To install the external environment module (Unix)

1. Locate the *php.ini* file for your PHP installation, and open it in a text editor. Locate the line that specifies the location of the **extension_dir** directory. If **extension_dir** is not set to any specific directory, it is a good idea to set it to point to an isolated directory for better system security.
2. Copy the desired PHP module from the SQL Anywhere installation directory to your PHP installation directory. The following is a model to use:

```
cp install-dir/bin32/php-5.2.6_sqlanywhere_extenv11.so
  php-dir/ext
```

Once the SQL Anywhere PHP external environment module has been built and installed, you can use the PHP in the database support. PHP in the database support is only available with SQL Anywhere version 11 or later databases. If a SQL Anywhere 10 database is loaded, then an error indicating that external environments are not supported is returned when you try to use the PHP in the database support.

To use PHP in the database, the database server must be able to locate and start the PHP executable. You can verify if the database server is able to locate and start the PHP executable by executing the following statement:

```
START EXTERNAL ENVIRONMENT PHP;
```

If the database server fails to start PHP, then the problem probably occurs because the database server is not able to locate the PHP executable. In this case, you should execute an ALTER EXTERNAL ENVIRONMENT statement to explicitly set the location of the PHP executable including the executable name.

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php-path';
```

For example:

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'c:\\php\\php-5.2.6-win32\\php.exe';
```

The START EXTERNAL ENVIRONMENT PHP statement is not necessary other than to verify that the database server can start PHP. In general, making a PHP stored procedure or function call starts PHP automatically.

Similarly, the STOP EXTERNAL ENVIRONMENT PHP statement is not necessary to stop an instance of PHP since the instance automatically goes away when the connection terminates. However, if you are completely done with PHP and you want to free up some resources, then the STOP EXTERNAL ENVIRONMENT PHP statement releases the PHP instance for your connection.

Once you have verified that the database server can start the PHP executable, the next thing to do is to install the necessary PHP code into the database. Do this by using the INSTALL statement. For example, you can execute the following statement to install a particular PHP script into the database.

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM FILE 'php-file'
  ENVIRONMENT PHP;
```

PHP code can also be built and installed from an expression as follows:

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;
```

PHP code can also be built and installed from a variable as follows:

```
CREATE VARIABLE PHPVariable LONG VARCHAR;
SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

To remove PHP code from the database, use the REMOVE statement as follows:

```
REMOVE EXTERNAL OBJECT 'php-script';
```

To modify existing PHP code, you can use the UPDATE clause of the INSTALL statement as follows:

```
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM FILE 'php-file'
  ENVIRONMENT PHP;

INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;

SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

Once the PHP code is installed in the database, you can then go ahead and create the necessary PHP stored procedures and functions. When creating PHP stored procedures and functions, the LANGUAGE is always PHP and the EXTERNAL NAME string contains the information needed to call the PHP subroutines and for returning OUT parameters.

The arguments are passed to the PHP script in the $argv array, similar to the way PHP would take arguments from the command line (that is, $argv[1] is the first argument). To set an output parameter, assign it to the appropriate $argv element. The return value is always the output from the script (as a LONG VARCHAR).

A PHP stored procedure can be created with any set of datatypes for input or output arguments. However, the parameters are converted to and from a boolean, integer, double, or string for use inside the PHP script. The return value is always an object of type LONG VARCHAR. A simple PHP example follows:

```
INSTALL EXTERNAL OBJECT 'SimplePHPExample'
  NEW
  FROM VALUE '<? function SimplePHPFunction(
    $arg1, $arg2, $arg3, $arg4 )
    { return ($arg1 * 1000) +
      ($arg2 * 100) +
      ($arg3 * 10) +
      $arg4;
    } ?>'
  ENVIRONMENT PHP;

CREATE FUNCTION SimplePHPDemo(
  IN thousands INT,
  IN hundreds INT,
  IN tens INT,
  IN ones INT)
RETURNS LONG VARCHAR
EXTERNAL NAME '<file=SimplePHPExample> print SimplePHPFunction(
    $argv[1], $argv[2], $argv[3], $argv[4]);'
LANGUAGE PHP;

// The number 1234 should appear
SELECT SimplePHPDemo(1,2,3,4);
```

For PHP, the EXTERNAL NAME string is specified in a single line of SQL.

To use server-side PHP, the PHP code can use the default database connection. To get a handle to the database connection, call sasql_pconnect with an empty string argument (" or ""). The empty string argument tells the SQL Anywhere PHP driver to return the current external environment connection rather than opening a new one. The following example creates a table and then calls a PHP stored procedure to populate the table:


```
CREATE TABLE phpTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<? function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    sasql_query( $conn,
    "INSERT INTO phpTab
       SELECT table_id, table_name FROM SYS.SYSTAB" );
    sasql_commit( $conn );
  } ?>'
  ENVIRONMENT PHP;

CREATE PROCEDURE PHPPopulateTable()
EXTERNAL NAME '<file=ServerSidePHPExample> ServerSidePHPSub()'
LANGUAGE PHP;

CALL PHPPopulateTable();

// The following should return 2 identical rows
SELECT count(*) FROM phpTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

For PHP, the EXTERNAL NAME string is specified in a single line of SQL. In the above example, note that the single quotes are doubled-up because of the way quotes are parsed in SQL. If the PHP source code was in a file, then the single quotes would not be doubled-up.

To return an error back to the database server, throw a PHP exception. The following example shows how to do this.

```
CREATE TABLE phpTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<? function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    if( !sasql_query( $conn,
      "INSERT INTO phpTabNoExist
         SELECT table_id, table_name FROM SYS.SYSTAB" )
    ) throw new Exception(
      sasql_error( $conn ),
      sasql_errorcode( $conn )
    );
    sasql_commit( $conn );
  } ?>'
  ENVIRONMENT PHP;

CREATE PROCEDURE PHPPopulateTable()
  EXTERNAL NAME
    '<file=ServerSidePHPExample> ServerSidePHPSub()'
  LANGUAGE PHP;

CALL PHPPopulateTable();
```

The above example should terminate with error SQLE_UNHANDLED_EXTENV_EXCEPTION indicating that the table `phpTabNoExist` could not be found.

For additional information and examples on using the PHP in the database support, refer to the examples located in the *samples-dir\SQLAnywhere\ExternalEnvironments\PHP* directory.

The ESQL and ODBC external environments

SQL Anywhere has had the ability to call compiled native functions written in C or C++ for some time. However, when these procedures are run by the database server, the dynamic link library or shared object has always been loaded by the database server and the calls out to the native functions have always been made by the database server. While having the database server make these native calls is most efficient, there can be serious consequences if the native function misbehaves. In particular, if the native function enters an infinite loop, then the database server can hang, and if the native function causes a fault, then the database server crashes. As a result, you now have the option of running compiled native functions outside of the database server, in an external environment. There are some key benefits to running a compiled native function in an external environment:

1. The database server does not hang or crash if the compiled native function misbehaves.
2. The native function can be written as either an ESQL or an ODBC application and can make server-side calls back into the database server without having to make a connection.
3. The native function can return a result set to the database server.
4. In the external environment, a 32-bit database server can communicate with a 64-bit compiled native function and vice versa. Note that this is not possible when the compiled native functions are loaded directly into the address space of the database server. A 32-bit library can only be loaded by a 32-bit server and a 64-bit library can only be loaded by a 64-bit server.

It should be noted, though, that running a compiled native function in an external environment instead of within the database server results in a small performance penalty.

It should also be noted that the compiled native function must use the native function call API. This API is described in "SQL Anywhere External Function API" on page 593.

To run a compiled native C function in an external environment instead of within the database server, the stored procedure or function is defined with the EXTERNAL NAME clause followed by the LANGUAGE clause specifying one of C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64.

Unlike the Perl, PHP, and Java external environments, the ESQL and ODBC environments do not have anything installed in the database. As a result, you do not need to execute any INSTALL statements prior to using the ESQL and ODBC external environments.

Here is an example of a function written in C++ that can be run within the database server or in an external environment.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}
```



```
// Note: extfn_use_new_api used only for
// execution in the database server

extern "C" __declspec( dllexport )
a_sql_uint32 extfn_use_new_api( void )
{
    return( EXTFN_API_VERSION );
}

extern "C" __declspec( dllexport )
void SimpleCFunction(
    an_extfn_api *api,
    void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    int *               intp;
    int                 i, j, k;

    j = 1000;
    k = 0;
    for( i = 1; i <= 4; i++ )
    {
        result = api->get_value( arg_handle, i, &arg );
        if( result == 0 || arg.data == NULL ) break;
        if( arg.type & DT_TYPES != DT_INT ) break;
        intp = (int *) arg.data;
        k += *intp * j;
        j = j / 10;
    }
    retval.type = DT_INT;
    retval.data = (void*)&k;
    retval.piece_len = retval.len.total_len =
       (a_sql_uint32) sizeof( int );
    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

When compiled into a dynamic link library or shared object, this function can be called from an external environment. An executable image called *dbexternc11* is started by the database server and this executable image loads the dynamic link library for you. Different versions of this executable are included with SQL Anywhere. For example, on Windows you may have both 32-bit and 64-bit executables.

Note that 32-bit or 64-bit versions of the database server can be used and either version can start 32-bit or 64-bit versions of *dbexternc11*. This is one of the advantages of using the external environment. Note that once *dbexternc11* is started by the database server, it does not terminate until the connection has been terminated.

In order to call the compiled native function, SimpleCFunction, a wrapper is defined as follows:

```
CREATE FUNCTION SimpleCDemo(
  IN arg1 INT,
  IN arg2 INT,
  IN arg3 INT,
  IN arg4 INT )
RETURNS INT
EXTERNAL NAME 'SimpleCFunction@d:\\c\\extdemo.dll'
LANGUAGE C_ODBC32;
```

This is almost identical to the way a compiled native function is described when it is to be loaded into the database server's address space. The one difference is the use of the LANGUAGE C_ODBC32 clause. This clause indicates that SimpleCDemo is a function running in an external environment and that it is using 32-bit ODBC calls.

To execute the sample compiled native function, execute the following statement.

```
SELECT SimpleCDemo(1,2,3,4);
```

The language specification of C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64 tells the database server whether the external C function issues 32-bit or 64-bit embedded SQL or ODBC calls when making server-side requests. If the function does not make any server-side requests, then either the ESQL or ODBC specification will do. For more information on how to make server-side requests and how to return result sets from an external C function, refer to the samples in *samples-dir\SQLAnywhere\ExternalEnvironments\ExternC*.

To use server-side ODBC, the C/C++ code must use the default database connection. To get a handle to the database connection, call get_value with an EXTFN_CONNECTION_HANDLE_ARG_NUM argument. The argument tells the database server to return the current external environment connection rather than opening a new one.

```
#include <windows.h>
#include <stdio.h>
#include "odbc.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    SQLRETURN           ret;

    ret = -1;
    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );

    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }

    HDBC dbc = (HDBC)arg.data;
```



```
    HSTMT stmt = SQL_NULL_HSTMT;
    ret = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
    if( ret != SQL_SUCCESS ) return;
    ret = SQLExecDirect( stmt,
            (SQLCHAR *) "INSERT INTO odbcTab "
                "SELECT table_id, table_name "
                "FROM SYS.SYSTAB", SQL_NTS );
    if( ret == SQL_SUCCESS )
    {
        SQLExecDirect( stmt,
            (SQLCHAR *) "COMMIT", SQL_NTS );
    }
    SQLFreeHandle( SQL_HANDLE_STMT, stmt );

    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

If the above ODBC code is stored in the file *extodbc.cpp*, it can be built for Windows using the following commands (assuming that the SQL Anywhere software is installed in the folder *c:\sa11* and that Microsoft Visual C++ is installed).

```
cl extodbc.cpp /LD /Ic:\sa11\sdk\include odbc32.lib
```

The following example creates a table, defines the stored procedure wrapper to call the compiled native function, and then calls the native function to populate the table.

```
CREATE TABLE odbcTab(c1 int, c2 char(128));

CREATE FUNCTION ServerSideODBC( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extodbc.dll'
LANGUAGE C_ODBC32;

SELECT ServerSideODBC();

// The following statement should return two identical rows
SELECT COUNT(*) FROM odbcTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

Similarly, to use server-side ESQL, the C/C++ code must use the default database connection. To get a handle to the database connection, call get_value with an EXTFN_CONNECTION_HANDLE_ARG_NUM argument. The argument tells the database server to return the current external environment connection rather than opening a new one.

```
#include <windows.h>
#include <stdio.h>

#include "sqlca.h"
#include "sqlda.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}
```

```
EXEC SQL INCLUDE SQLCA;
static SQLCA *_sqlc;
EXEC SQL SET SQLCA "_sqlc";
EXEC SQL WHENEVER SQLERROR { ret = _sqlc->sqlcode; };

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;

    EXEC SQL BEGIN DECLARE SECTION;
    char *stmt_text =
        "INSERT INTO esqlTab "
            "SELECT table_id, table_name "
            "FROM SYS.SYSTAB";
    char *stmt_commit =
        "COMMIT";
    EXEC SQL END DECLARE SECTION;

    int ret = -1;

    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );

    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
    ret = 0;
    _sqlc = (SQLCA *)arg.data;

    EXEC SQL EXECUTE IMMEDIATE :stmt_text;
    EXEC SQL EXECUTE IMMEDIATE :stmt_commit;

    api->set_value( arg_handle, 0, &retval, 0 );
}
```

If the above embedded SQL code is stored in the file *extesql.sqc*, it can be built for Windows using the following commands (assuming that the SQL Anywhere software is installed in the folder *c:\sa11* and that Microsoft Visual C++ is installed).

```
sqlpp extesql.sqc extesql.cpp
cl extesql.cpp /LD /Ic:\sa11\sdk\include c:\sa11\sdk\lib\x86\dblibtm.lib
```

The following example creates a table, defines the stored procedure wrapper to call the compiled native function, and then calls the native function to populate the table.

```
CREATE TABLE esqlTab(c1 int, c2 char(128));

CREATE FUNCTION ServerSideESQL( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extesql.dll'
LANGUAGE C_ESQL32;
```



```
SELECT ServerSideESQL();

// The following statement should return two identical rows
SELECT COUNT(*) FROM esqlTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

For additional information, see "SQL Anywhere External Function API" on page 593.

The CLR external environment

SQL Anywhere includes support for CLR stored procedures and functions. A CLR stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in C# or Visual Basic, and the execution of the procedure or function takes place outside of the database server (that is, within a separate .NET executable). There is only one instance of this .NET executable per database. All connections executing CLR functions and stored procedures use the same .NET executable instance, but the namespaces for each connection are separate. Statics persist for the duration of the connection, but are not shareable across connections. Only .NET version 2.0 is supported.

To call an external CLR function or procedure, you define a corresponding stored procedure or function with an EXTERNAL NAME string defining which DLL to load and which function within the assembly to call. You must also specify LANGUAGE CLR when defining the stored procedure or function. An example declaration follows:

```
CREATE PROCEDURE clr_stored_proc(
    IN p1 INT,
    IN p2 UNSIGNED SMALLINT,
    OUT p3 LONG VARCHAR)
EXTERNAL NAME 'MyCLRTest.dll::MyCLRTest.Run( int, ushort, out string )'
LANGUAGE CLR;
```

In this example, the stored procedure called `clr_stored_proc`, when executed, loads the DLL *MyCLRTest.dll* and calls the function MyCLRTest.Run. The clr_stored_proc procedure takes three SQL parameters, two IN parameters, one of type INT and one of type UNSIGNED SMALLINT, and one OUT parameter of type LONG VARCHAR. On the .NET side, these three parameters translate to input arguments of type int and ushort and an output argument of type string. In addition to out arguments, the CLR function can also have ref arguments. A user must declared a ref CLR argument if the corresponding stored procedure has an INOUT parameter.

The following table lists the various CLR argument types and the corresponding suggested SQL datatype:

CLR type	Recommended SQL data type
bool	bit
byte	tinyint
short	smallint
ushort	unsigned smallint
int	int
uint	unsigned int
long	bigint
ulong	unsigned bigint

CLR type	Recommended SQL data type
decimal	numeric
float	real
double	double
DateTime	timestamp
string	long varchar
byte[]	long binary

The declaration of the DLL can be either a relative or absolute path. If the specified path is relative, then the external .NET executable searches the path, as well as other places, for the DLL. The executable does not search the Global Assembly Cache (GAC) for the DLL.

Like the existing Java stored procedures and functions, CLR stored procedures and functions can make server-side requests back to the database, and they can return result sets. Also, like Java, any information output to Console.Out and Console.Error is automatically redirected to the database server messages window.

For more information about how to make server-side requests and how to return result sets from a CLR function or stored procedure, refer to the samples located in the *samples-dir\SQLAnywhere \ExternalEnvironments\CLR* directory.

To use CLR in the database, make sure the database server is able to locate and start the CLR executable. You can verify if the database server is able to locate and start the CLR executable by executing the following statement:

```
START EXTERNAL ENVIRONMENT CLR;
```

If the database server fails to start CLR, then the database server is likely not able to locate the CLR executable. The CLR executable is *dbextclr11.exe*. Make sure that this file is present in the *install-dir \Bin32* or *install-dir\Bin64* folder, depending on which version of the database server you are using.

Note that the START EXTERNAL ENVIRONMENT CLR statement is not necessary other than to verify that the database server can launch CLR executables. In general, making a CLR stored procedure or function call starts CLR automatically.

Similarly, the STOP EXTERNAL ENVIRONMENT CLR statement is not necessary to stop an instance of CLR since the instance automatically goes away when the connection terminates. However, if you are completely done with CLR and you want to free up some resources, then the STOP EXTERNAL ENVIRONMENT CLR statement releases the CLR instance for your connection.

Unlike the Perl, PHP, and Java external environments, the CLR environment does not require the installation of anything in the database. As a result, you do not need to execute any INSTALL statements prior to using of the CLR external environment.

Here is an example of a function written in C# that can be run within an external environment.

```
public class StaticTest
{
    private static int val = 0;

    public static int GetValue() {
        val += 1;
        return val;
    }
}
```

When compiled into a dynamic link library, this function can be called from an external environment. An executable image called *dbextclr11.exe* is started by the database server and it loads the dynamic link library for you. Different versions of this executable are included with SQL Anywhere. For example, on Windows you may have both 32-bit and 64-bit executables. One is for use with the 32-bit version of the database server and the other for the 64-bit version of the database server.

To build this application into a dynamic link library using Microsoft's C# compiler, use a command like the following. The source code for the above example is assumed to reside in a file called *StaticTest.cs*.

```
csc /target:library /out:clrtest.dll StaticTest.cs
```

This command places the compiled code in a DLL called *clrtest.dll*. To call the compiled C# function, GetValue, a wrapper is defined as follows using Interactive SQL:

```
CREATE FUNCTION stc_get_value()
RETURNS INT
EXTERNAL NAME 'clrtest.dll::StaticTest.GetValue() int'
LANGUAGE CLR;
```

For CLR, the EXTERNAL NAME string is specified in a single line of SQL. You may be required to include the path to the DLL as part of the EXTERNAL NAME string so that it can be located.

To execute the sample compiled C# function, execute the following statement.

```
SELECT stc_get_value();
```

Each time the C# function is called, a new integer result is produced. The sequence of values returned is 1, 2, 3, and so on.

For additional information and examples on using the CLR in the database support, refer to the examples located in the *samples-dir\SQLAnywhere\ExternalEnvironments\CLR* directory.

CHAPTER 20

Sybase Open Client API

Contents

Open Client architecture

> **Note**
> This chapter describes the Sybase Open Client programming interface for SQL Anywhere. The primary documentation for Sybase Open Client application development is the Open Client documentation, available from Sybase. This chapter describes features specific to SQL Anywhere, but it is not an exhaustive guide to Sybase Open Client application programming.

Sybase Open Client has two components: programming interfaces and network services.

DB-Library and Client Library

Sybase Open Client provides two core programming interfaces for writing client applications: DB-Library and Client-Library.

Open Client DB-Library provides support for older Open Client applications, and is a completely separate programming interface from Client-Library. DB-Library is documented in the *Open Client DB-Library/C Reference Manual*, provided with the Sybase Open Client product.

Client-Library programs also depend on CS-Library, which provides routines that are used in both Client-Library and Server-Library applications. Client-Library applications can also use routines from Bulk-Library to facilitate high-speed data transfer.

Both CS-Library and Bulk-Library are included in the Sybase Open Client, which is available separately.

Network services

Open Client network services include Sybase Net-Library, which provides support for specific network protocols such as TCP/IP and DECnet. The Net-Library interface is invisible to application programmers. However, on some platforms, an application may need a different Net-Library driver for different system network configurations. Depending on your host platform, the Net-Library driver is specified either by the system's Sybase configuration or when you compile and link your programs.

Instructions for driver configuration can be found in the *Open Client/Server Configuration Guide*.

Instructions for building Client-Library programs can be found in the *Open Client/Server Programmer's Supplement*.

What you need to build Open Client applications

To run Open Client applications, you must install and configure Sybase Open Client components on the computer where the application is running. You may have these components present as part of your installation of other Sybase products or you can optionally install these libraries with SQL Anywhere, subject to the terms of your license agreement.

Open Client applications do not need any Open Client components on the computer where the database server is running.

To build Open Client applications, you need the development version of Open Client, available from Sybase.

By default, SQL Anywhere databases are created as case-insensitive, while Adaptive Server Enterprise databases are case sensitive.

For more information about running Open Client applications with SQL Anywhere, see "Using SQL Anywhere as an Open Server" [*SQL Anywhere Server - Database Administration*].

Data type mappings

Sybase Open Client has its own internal data types, which differ in some details from those available in SQL Anywhere. For this reason, SQL Anywhere internally maps some data types between those used by Open Client applications and those available in SQL Anywhere.

To build Open Client applications, you need the development version of Open Client. To use Open Client applications, the Open Client runtimes must be installed and configured on the computer where the application runs.

The SQL Anywhere server does not require any external communications runtime to support Open Client applications.

Each Open Client data type is mapped onto the equivalent SQL Anywhere data type. All Open Client data types are supported

SQL Anywhere data types with no direct counterpart in Open Client

The following table lists the mappings of data types supported in SQL Anywhere that have no direct counterpart in Open Client.

SQL Anywhere data type	Open Client data type
unsigned short	int
unsigned int	bigint
unsigned bigint	numeric(20,0)
date	smalldatetime
time	smalldatetime
string	varchar
timestamp	datetime

Range limitations in data type mapping

Some data types have different ranges in SQL Anywhere than in Open Client. In such cases, overflow errors can occur during retrieval or insertion of data.

The following table lists Open Client application data types that can be mapped to SQL Anywhere data types, but with some restriction in the range of possible values.

In most cases, the Open Client data type is mapped to a SQL Anywhere data type that has a greater range of possible values. As a result, it is possible to pass a value to SQL Anywhere that will be accepted and stored in a database, but that is too large to be fetched by an Open Client application.

Data type	Open Client lower range	Open Client upper range	SQL Anywhere lower range	SQL Anywhere upper range
MONEY	-922 377 203 685 477.5808	922 377 203 685 477.5807	-1e15 + 0.0001	1e15 - 0.0001
SMALLMONEY	-214 748.3648	214 748.3647	-214 748.3648	214 748.3647
DATETIME	Jan 1, 1753	Dec 31, 9999	Jan 1, 0001	Dec 31, 9999
SMALLDATETIME	Jan 1, 1900	June 6, 2079	March 1, 1600	Dec 31, 7910

Example

For example, the Open Client MONEY and SMALLMONEY data types do not span the entire numeric range of their underlying SQL Anywhere implementations. Therefore, it is possible to have a value in a SQL Anywhere column which exceeds the boundaries of the Open Client data type MONEY. When the client fetches any such offending values via SQL Anywhere, an error is generated.

Timestamps

The SQL Anywhere implementation of the Open Client TIMESTAMP data type, when such a value is passed in SQL Anywhere, is different from that of Adaptive Server Enterprise. In SQL Anywhere, the value is mapped to the SQL Anywhere DATETIME data type. The default value is NULL in SQL Anywhere and no guarantee is made of its uniqueness. By contrast, Adaptive Server Enterprise ensures that the value is monotonically increasing in value, and so, is unique.

By contrast, the SQL Anywhere TIMESTAMP data type contains year, month, day, hour, minute, second, and fraction of second information. In addition, the DATETIME data type has a greater range of possible values than the Open Client data types that are mapped to it by SQL Anywhere.

Using SQL in Open Client applications

This section provides a very brief introduction to using SQL in Open Client applications, with a particular focus on SQL Anywhere-specific issues.

For an introduction to the concepts, see "Using SQL in applications" on page 21. For a complete description, see your Open Client documentation.

Executing SQL statements

You send SQL statements to a database by including them in Client Library function calls. For example, the following pair of calls executes a DELETE statement:

```
ret = ct_command(cmd, CS_LANG_CMD,
                  "DELETE FROM Employees
                   WHERE EmployeeID=105"
                 CS_NULLTERM,
                 CS_UNUSED);
ret = ct_send(cmd);
```

For more information on Open Client functions, see Open Client 15.0 Client-Library/C Reference Manual.

Using prepared statements

The ct_dynamic function is used to manage prepared statements. This function takes a *type* parameter that describes the action you are taking.

To use a prepared statement in Open Client

1. Prepare the statement using the ct_dynamic function, with a CS_PREPARE *type* parameter.
2. Set statement parameters using ct_param.
3. Execute the statement using ct_dynamic with a CS_EXECUTE *type* parameter.
4. Free the resources associated with the statement using ct_dynamic with a CS_DEALLOC *type* parameter.

For more information about using prepared statements in Open Client, see your Open Client documentation

Using cursors

The ct_cursor function is used to manage cursors. This function takes a *type* parameter that describes the action you are taking.

Supported cursor types

Not all the types of cursor that SQL Anywhere supports are available through the Open Client interface. You cannot use scroll cursors, dynamic scroll cursors, or insensitive cursors through Open Client.

Uniqueness and updatability are two properties of cursors. Cursors can be unique (each row carries primary key or uniqueness information, regardless of whether it is used by the application) or not. Cursors can be read only or updatable. If a cursor is updatable and not unique, performance may suffer, as no prefetching of rows is done in this case, regardless of the CS_CURSOR_ROWS setting.

The steps in using cursors

In contrast to some other interfaces, such as embedded SQL, Open Client associates a cursor with a SQL statement expressed as a string. Embedded SQL first prepares a statement and then the cursor is declared using the statement handle.

To use cursors in Open Client

1. To declare a cursor in Open Client, use ct_cursor with CS_CURSOR_DECLARE as the *type* parameter.
2. After declaring a cursor, you can control how many rows are prefetched to the client side each time a row is fetched from the server by using ct_cursor with CS_CURSOR_ROWS as the *type* parameter.

   Storing prefetched rows at the client side reduces the number of calls to the server and this improves overall throughput, as well as turnaround time. Prefetched rows are not immediately passed on to the application; they are stored in a buffer at the client side ready for use.

   The setting of the prefetch database option controls prefetching of rows for other interfaces. It is ignored by Open Client connections. The CS_CURSOR_ROWS setting is ignored for non-unique, updatable cursors.
3. To open a cursor in Open Client, use ct_cursor with CS_CURSOR_OPEN as the *type* parameter.
4. To fetch each row in to the application, use ct_fetch.
5. To close a cursor, you use ct_cursor with CS_CURSOR_CLOSE.
6. In Open Client, you also need to deallocate the resources associated with a cursor. You do this by using ct_cursor with CS_CURSOR_DEALLOC. You can also use CS_CURSOR_CLOSE with the additional parameter CS_DEALLOC to perform these operations in a single step.

Modifying rows through a cursor

With Open Client, you can delete or update rows in a cursor, as long as the cursor is for a single table. The user must have permissions to update the table and the cursor must be marked for update.

To modify rows through a cursor

- Instead of carrying out a fetch, you can delete or update the current row of the cursor using ct_cursor with CS_CURSOR_DELETE or CS_CURSOR_UPDATE, respectively.

You cannot insert rows through a cursor in Open Client applications.

Describing query results in Open Client

Open Client handles result sets in a different way than some other SQL Anywhere interfaces.

In embedded SQL and ODBC, you **describe** a query or stored procedure to set up the proper number and types of variables to receive the results. The description is done on the statement itself.

In Open Client, you do not need to describe a statement. Instead, each row returned from the server can carry a description of its contents. If you use ct_command and ct_send to execute statements, you can use the ct_results function to handle all aspects of rows returned in queries.

If you do not want to use this row-by-row method of handling result sets, you can use ct_dynamic to prepare a SQL statement and use ct_describe to describe its result set. This corresponds more closely to the describing of SQL statements in other interfaces.

Known Open Client limitations of SQL Anywhere

Using the Open Client interface, you can use a SQL Anywhere database in much the same way as you would an Adaptive Server Enterprise database. There are some limitations, including the following:

- SQL Anywhere does not support the Adaptive Server Enterprise Commit Service.
- A client/server connection's **capabilities** determine the types of client requests and server responses permitted for that connection. The following capabilities are not supported:
  - CS_CSR_ABS
  - CS_CSR_FIRST
  - CS_CSR_LAST
  - CS_CSR_PREV
  - CS_CSR_REL
  - CS_DATA_BOUNDARY
  - CS_DATA_SENSITIVITY
  - CS_OPT_FORMATONLY
  - CS_PROTO_DYNPROC
  - CS_REG_NOTIF
  - CS_REQ_BCP
- Security options, such as SSL, are not supported. However, password encryption is supported.
- Open Client applications can connect to SQL Anywhere using TCP/IP.

  For more information about capabilities, see the *Open Server Server-Library C Reference Manual*.
- When the CS_DATAFMT is used with the CS_DESCRIBE_INPUT, it does not return the data type of a column when a parameterized variable is sent to SQL Anywhere as input.

CHAPTER 21

SQL Anywhere web services

Contents

Introduction to web services

SQL Anywhere contains a built-in HTTP server that allows you to provide web services, as well as to access web services in other SQL Anywhere databases and standard web services available over the Internet. SOAP is the standard used for this purpose, but the built-in HTTP server in SQL Anywhere also lets you handle standard HTTP and HTTPS requests from client applications.

The term web service has been used to mean a variety of things. Commonly, it refers to software that facilitates inter-computer data transfer and interoperability. Essentially, web services make segments of business logic available over the Internet. **Simple Object Access Protocol** (SOAP) is a simple XML-based protocol to let applications exchange information over HTTP.

SOAP provides a method for communication, using the Internet, between applications such as those written in Java or a Microsoft .NET language like Visual C#. SOAP messages define the services that a server provides. Actual data transfer generally takes place using HTTP to exchange XML documents structured so as to efficiently encode relevant information. Any application, such as a client or server, that participates in SOAP communication is called a SOAP node or SOAP endpoint. Such applications can transmit, receive, or process SOAP messages. You can create SOAP nodes with SQL Anywhere.

For more information about the SOAP standards, see www.w3.org/TR/soap.

Web services and SQL Anywhere

In the context of SQL Anywhere, the term web services means that SQL Anywhere has the ability to listen for and handle standard SOAP requests. Web services in SQL Anywhere provide client applications an alternative to such traditional interfaces as JDBC and ODBC. Web services can be accessed from client applications written in a variety of languages and running on a variety of platforms. Even common scripting languages such as Perl and Python provide access to web services. You create web services in a database using the CREATE SERVICE statement.

SQL Anywhere can also function as a SOAP or HTTP client, permitting applications running within the database to access standard web services available over the Internet, or provided by other SQL Anywhere databases. This client functionality is accessed through stored functions and procedures.

In addition, the term web services also refers to applications that use the built-in web server to handle HTTP requests from clients. These applications generally function like traditional database-backed web applications, but can be more compact and are easier to write as the data and the entire application can reside within a database. In this type of application, the web service typically returns documents in HTML format. The GET, HEAD, and POST methods are supported.

The collection of web services within your database together define the available URLs. Each service provides a set of web pages. Typically, the content of these pages is generated by procedures that you write and store in your database, although they can be a single statement or, optionally, allow users to execute statements of their own. These web services become available when you start the database server with options that enable it to listen for HTTP requests.

Since the HTTP server that handles web service requests is embedded in the database, performance is good. Applications that use web services are easily deployed, since no additional components are needed, beyond the database and database server.

Quick start to web services

The following procedure describes how to create a new database, start a SQL Anywhere database with the HTTP server enabled, and access this database using any popular web browser.

To create and access a simple HTML web service

1. Execute the following statement to start a personal web server. The `-xs http(port=80)` option tells the database server to listen for HTTP requests. If you already have a web server running on port 80, use another port number such as 8080 for this demonstration.

   ```
   dbeng11 -xs http(port=80) c:\webserver\demo.db
   ```

   Many properties of the HTTP communication link are controlled by parameters to the -xs option.

   For more information, see "-xs server option" [*SQL Anywhere Server - Database Administration*].

2. As well as starting the database server with the appropriate -xs option parameters, you must create web services to respond to incoming requests. These are defined using the CREATE SERVICE statement.

   Start Interactive SQL. Connect to the SQL Anywhere sample database as the DBA. Execute the following statement.

   ```
   CREATE SERVICE HTMLtable
   TYPE 'HTML'
   AUTHORIZATION OFF
   USER DBA
   AS SELECT * FROM Customers;
   ```

   This statement creates a web service named HTMLtable. This simple web service returns the results of the statement SELECT * FROM Customers, automatically converting the output into HTML format. Because authorization is off, no permissions are required to access the table from a web browser.

   For more information, see "Creating web services" on page 722 and "CREATE SERVICE statement" [*SQL Anywhere Server - SQL Reference*].

3. Start a web browser.

4. Browse to the URL http://localhost:80/demo/HTMLtable. Use the port number you specified when starting the database server.

   Your web browser shows you the body of the HTML document returned by the database server. By default, the result set is formatted into an HTML table.

To create and access a simple XML web service

1. Copy the SQL Anywhere sample database from *samples-dir* to another location, such as *c:\webserver\demo.db*.

2. Execute the following statement to start a personal web server. The `-xs http(port=80)` option tells the database server to listen for HTTP requests. If you already have a web server running on port 80, use another port number such as 8080 for this demonstration.

   ```
   dbeng11 -xs http(port=80) c:\webserver\demo.db
   ```

   Many properties of the HTTP communication link are controlled by parameters to the -xs option.

For more information, see "-xs server option" [*SQL Anywhere Server - Database Administration*].

3. As well as starting the database server with the appropriate -xs option parameters, you must create web services to respond to incoming requests. These are defined using the CREATE SERVICE statement.

   Start Interactive SQL. Connect to the SQL Anywhere sample database as the DBA. Execute the following statement:

   ```
   CREATE SERVICE XMLtable
   TYPE 'XML'
   AUTHORIZATION OFF
   USER DBA
   AS SELECT * FROM Customers;
   ```

   This statement creates a web service named XMLtable. This simple web service returns the results of the statement SELECT * FROM Customers, automatically converting the output into XML format. Because authorization is off, no permissions are required to access the table from a web browser.

   For more information, see "Creating web services" on page 722 and "CREATE SERVICE statement" [*SQL Anywhere Server - SQL Reference*].

4. Start a web browser.

5. Browse to the URL http://localhost:80/demo/XMLtable. Use the port number you specified when starting the database server.

   - **localhost:80** Defines the web host name and port number to use.
   - **demo** Defines the database name to use. You are using *c:\webserver\demo.db*.
   - **XMLtable** Defines the service name to use.

   Your web browser shows you the body of the XML document returned by the database server. As no formatting information has been included, you see the raw XML, including tags and attributes.

6. You can also access the XMLtable service from common programming languages. For example, the following short C# program uses the XMLtable web service:

   ```
   using System.Xml;

   static void Main(string[] args)
   {
     XmlTextReader reader =
       new XmlTextReader( "http://localhost:80/demo/XMLtable" );

     while( reader.Read() )
     {
       switch( reader.NodeType )
       {
       case XmlNodeType.Element:
         if( reader.Name == "row" )
         {
           Console.Write(reader.GetAttribute("ID")+" ");
           Console.WriteLine(reader.GetAttribute("Surname"));
         }
         break;
       }
     }
     reader.Close();
   }
   ```


7. In addition, you can access the same web service from Python, as in the following example:

```
import xml.sax

class DocHandler( xml.sax.ContentHandler ):
  def startElement( self, name, attrs ):
    if name == 'row':
      table_id = attrs.getValue( 'ID' )
      table_name = attrs.getValue( 'Surname' )
      print '%s %s' % ( table_id, table_name )

parser = xml.sax.make_parser()
parser.setContentHandler( DocHandler() )
parser.parse( 'http://localhost:80/demo/XMLtable' )
```

   Save this code in a file called *DocHandler.py*. To run the application, enter a command like the following:

```
python DocHandler.py
```

To create and access a simple JSON web service

1. Copy the SQL Anywhere sample database from *samples-dir* to another location, such as *c:\webserver\demo.db*.
2. Execute the following statement to start a personal web server. The `-xs http(port=80)` option tells the database server to listen for HTTP requests. If you already have a web server running on port 80, use another port number such as 8080 for this demonstration.

```
dbeng11 -xs http(port=80) c:\webserver\demo.db
```

   Many properties of the HTTP communication link are controlled by parameters to the -xs option.

   For more information, see "-xs server option" [*SQL Anywhere Server - Database Administration*].
3. As well as starting the database server with the appropriate -xs option parameters, you must create web services to respond to incoming requests. These are defined using the CREATE SERVICE statement.

   Start Interactive SQL. Connect to the SQL Anywhere sample database as the DBA. Execute the following statement:

```
CREATE SERVICE JSONtable
TYPE 'JSON'
AUTHORIZATION OFF
USER DBA
AS SELECT * FROM Customers;
```

   This statement creates a web service named JSONtable. This simple web service returns the results of the statement SELECT * FROM Customers, automatically converting the output into JavaScript Object Notation format. Because authorization is off, no permissions are required to access the table from a web browser.

   For more information, see "Creating web services" on page 722 and "CREATE SERVICE statement" [*SQL Anywhere Server - SQL Reference*].
4. Start a web browser.
5. Browse to the URL http://localhost:80/demo/JSONtable. Use the port number you specified when starting the database server.

- **localhost:80** Defines the web host name and port number to use.
- **demo** Defines the database name to use. You are using *c:\webserver\demo.db*.
- **JSONtable** Defines the service name to use.

Your web browser should permit you to save the JSON response document returned by the database server. Save the response to a file.

6. If you use a text editor to view the file containing the response, you will see the following array notation for the result set.

```
[
        {
                "ID": 101,
                "Surname": "Devlin",
                "GivenName": "Michaels",
                "Street": "114 Pioneer Avenue",
                "City": "Kingston",
                "State": "NJ",
                "Country": "USA",
                "PostalCode": "07070",
                "Phone": "2015558966",
                "CompanyName": "The Power Group"
        },
        {
                "ID": 102,
                "Surname": "Reiser",
                "GivenName": "Beth",
                "Street": "33 Whippany Road",
                "City": "Rockwood",
                "State": "NY",
                "Country": "USA",
                "PostalCode": "10154",
                "Phone": "2125558725",
                "CompanyName": "AMF Corp."
        },
.
.
.
        {
                "ID": 665,
                "Surname": "Thompson",
                "GivenName": "William",
                "Street": "19 Washington Street",
                "City": "Bancroft",
                "State": "NY",
                "Country": "USA",
                "PostalCode": "11700",
                "Phone": "5165552549",
                "CompanyName": "The Apple Farm"
        }
]
```

Other resources for getting started

Samples are included in the *samples-dir\SQLAnywhere\HTTP* directory.

Other examples might be available on CodeXchange at http://ianywhere.codexchange.sybase.com/.

See also

- "HTML_DECODE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]

- "HTML_ENCODE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "HTTP_DECODE function [HTTP]" [*SQL Anywhere Server - SQL Reference*]
- "HTTP_ENCODE function [HTTP]" [*SQL Anywhere Server - SQL Reference*]
- "HTTP_HEADER function [HTTP]" [*SQL Anywhere Server - SQL Reference*]
- "HTTP_VARIABLE function [HTTP]" [*SQL Anywhere Server - SQL Reference*]
- "NEXT_HTTP_HEADER function [HTTP]" [*SQL Anywhere Server - SQL Reference*]
- "NEXT_HTTP_VARIABLE function [HTTP]" [*SQL Anywhere Server - SQL Reference*]
- "NEXT_SOAP_HEADER function [SOAP]" [*SQL Anywhere Server - SQL Reference*]
- "SOAP_HEADER function [SOAP]" [*SQL Anywhere Server - SQL Reference*]
- "sa_http_header_info system procedure" [*SQL Anywhere Server - SQL Reference*]
- "sa_http_variable_info system procedure" [*SQL Anywhere Server - SQL Reference*]
- "sa_set_http_header system procedure" [*SQL Anywhere Server - SQL Reference*]
- "sa_set_http_option system procedure" [*SQL Anywhere Server - SQL Reference*]
- "sa_set_soap_header system procedure" [*SQL Anywhere Server - SQL Reference*]

Creating web services

Web services, created and stored in databases, define which URLs are valid and what they do. A single database can define multiple web services. It is possible to define web services in different databases so that they appear to be part of a single web site.

The following statements permit you to create, alter, and delete web services:

- CREATE SERVICE
- ALTER SERVICE
- DROP SERVICE
- COMMENT ON SERVICE

The general syntax of the CREATE SERVICE statement is as follows:

CREATE SERVICE *service-name* **TYPE '***service-type***'** [*attributes*] [**AS** *statement*]

Service names

Since service names form part of the URL used to access them, they are flexible in terms of what characters they can contain. In addition to the standard alpha-numeric characters, the following characters are permitted:
`- _ . ! * '( )`

In addition, service names other than those used in naming DISH services can contain a slash, "/", but some restrictions apply because this character is a standard URL delimiter and affects how SQL Anywhere interprets your URLs. It cannot be the first character of a service name. In addition, service names cannot contain two consecutive slashes.

The characters permitted in service names are also permitted in GROUP names, which apply to DISH services only.

Service types

The following service types are supported:

- **'SOAP'** The result set is returned as a SOAP response. The format of the data is determined by the FORMAT clause. A request to a SOAP service must be a valid SOAP request, not just a simple HTTP request.
- **'DISH'** A DISH service (Determine SOAP Handler) acts as a proxy for those SOAP services identified by the GROUP clause, and generates a WSDL (Web Services Description Language) document for each of these SOAP services.
- **'HTML'** The result set of a statement or procedure is automatically formatted into an HTML document that contains a table.
- **'XML'** The result set is returned as XML. If the result set is already XML, no additional formatting is applied. If it is not already XML, it is automatically formatted as XML. The effect is similar to that of using the FOR XML RAW clause in a SELECT statement.
- **'JSON'** The result set is returned in JavaScript Object Notation (JSON). JSON is more compact than XML and has a similar structure. For more information about JSON, visit http://www.json.org.

- **'RAW'** The result set is sent to the client without any further formatting. You can produce formatted documents by generating the required tags explicitly within your procedure.

Of all the service types, RAW gives you the most control over the output. However, it does require that you do more work as you must explicitly output all the necessary tags. The output of XML services can be adjusted by applying the FOR XML clause to the service's statement. The output of SOAP services can be adjusted using the FORMAT attribute of the CREATE or ALTER SERVICE statement.

For more information, see "CREATE SERVICE statement" [*SQL Anywhere Server - SQL Reference*].

Statements

The statement is the command, usually a stored procedure, that is called when someone accesses the service. If you define a statement, this is the only statement that can be run through this service. The statement is mandatory for SOAP services, and ignored for DISH services. The default is NULL, which means no statement.

You can create services that do not include statements. The statement is taken from the URL. Services configured in this way can be useful when you are testing a service, or want a general way of accessing information. To do so, either omit the statement entirely or use the phrase AS NULL in place of the statement.

Services without statements are a serious security risk because they permit web clients to execute arbitrary commands. When creating such services, you must enable authorization, which forces all clients to provide a valid user name and password. Even so, only services that define statements should be run in a production system.

Attributes

In general, all attributes are optional. However, some are interdependent. The following attributes are available:

- **AUTHORIZATION** This attribute controls which users can use the service. The default setting is ON. Authorization must be ON if no statement is provided. In addition, the authorization setting affects how user names, defined by the USER attribute, are interpreted.
- **SECURE** When set to ON, only secure connections are permitted. All connections received on the HTTP port are automatically redirected to the HTTPS port. The default is OFF, which enables both HTTP and HTTPS requests, provided these ports are enabled using the appropriate options when the database server is started.

  For more information, see "-xs server option" [*SQL Anywhere Server - Database Administration*].
- **USER** The USER clause controls which database user accounts can be used to process service requests. However, the interpretation of this setting depends on whether authorization is ON or OFF.

  When authorization is set to ON, all clients must provide a valid user name and password when they connect. When authorization is ON, the USER option can be NULL, a database user name, or the name of a database group. If it is NULL, any database user can connect and make requests. Requests are run using the account and permissions of that user. If a group name is specified, only those users who belong to the group can run requests. All other database users are denied permission to use the service.

  If authorization is OFF, a statement must be provided. In addition, a user name must be provided. All requests are run using that user's account and permissions. Thus, if the server is connected to a public

network, the permissions of the named user account should be minimal to limit the damage that could be caused through malicious use.

- **GROUP** The GROUP clause, which applies to DISH services only, determines which SOAP services are exposed by the DISH service. Only SOAP services whose names begin with the name of the group name of a DISH service are exposed by that DISH service. Thus, the group name is a common prefix among the exposed SOAP services. For example, specifying GROUP *xyz* exposes only SOAP services *xyz/aaaa*, *xyz/bbbb*, or *xyz/cccc*, but does not expose *abc/aaaa* or *xyzaaaa*. If no group name is specified, the DISH service exposes all the SOAP services in the database. The same characters are permitted in group names as in service names.

  SOAP services can be exposed by more than one DISH service. In particular, this feature permits a single SOAP service to supply data in multiple formats. The service type, unless specified in a SOAP service, is inherited from the DISH service. Thus, you can create a SOAP service that declares no format type, then include it in multiple DISH services, each of which specifies a different format.

- **FORMAT** The FORMAT clause, which applies to DISH and SOAP services only, controls the output format of the SOAP or DISH response. Output formats compatible with various types of SOAP clients, such as .NET or JAX-WS, are available. If the format of a SOAP service is not specified, the format is inherited from the service's DISH service declaration. If the DISH service also does not declare a format, it defaults to DNET, which is compatible with .NET clients. A SOAP service that does not declare a format can be used with different types of SOAP clients by defining multiple DISH services, each having a different FORMAT type.

- **URL [PATH]** The URL or URL PATH clause controls the interpretation of URLs and applies to XML, HTML, and RAW service types only. In particular, it determines whether URL paths are accepted and, if so, how they are processed. If the service name ends with the character "/", URL must be set to OFF.

  For more information, see "CREATE SERVICE statement" [*SQL Anywhere Server - SQL Reference*].

Starting a database server that listens for web requests

When you want a database server to listen for web service requests over HTTP or HTTPS, you must specify the types of web requests it is to listen on the command line when you start the server. By default, database servers do not listen for web service requests, leaving no way for clients to access any services that may be defined in your database.

You can also specify various properties of an HTTP or HTTPS service on the command line, such as on which port they are to listen.

You must also create web services within the database. For more information, see "Creating web services" on page 722.

You use the -xs option to enable protocols. The two available web service protocols are HTTP and HTTPS. Optional parameters, placed within parentheses after the protocol name, let you customize access to each type of web service.

The general syntax of the option is as follows:

-xs { *protocol* [**(***option***=***value*; ...**)**], ... }

Starting multiple web servers
If you want to start multiple web servers at the same time, then you must change the port for additional web servers since they all have the same default port.

Protocols

The following web service protocol values are available:

- **http** Listen for HTTP connections.
- **https** Listen for HTTPS connections. HTTPS connections using SSL version 3.0 and TLS version 1.0 are supported.
- **none** Do not listen for web service requests. This is the default setting.

Options

The following are some of the options that are available:

- **FIPS** Specify **FIPS=Y** to listen for HTTPS FIPS connections.
- **ServerPort [PORT]** The port on which to listen for web requests. By default, SQL Anywhere listens on port 80 for HTTP requests and on port 443 for secure HTTP (HTTPS) requests. The default port for FIPS-approved HTTPS connections is the same as for HTTPS.

  For example, if you already have a web server running on port 80, you could use the following options to start a database server that listens for web requests on port 8080:

  ```
  dbeng11 mywebapp.db -xs http(port=8080)
  ```

As another example, the following command starts a secure web server using the sample identity file included with SQL Anywhere (you must have installed RSA or FIPS-approved RSA encryption to have this file). It should be entered on a single line.

```
dbeng11 -xs https(identity=rsaserver.id;
                  identity_password=test)
```

Caution
The sample identity file is intended for use only during testing and development. It provides no protection because it is a standard part of SQL Anywhere. Replace it with your own certificate before deploying your application.

- **DatabaseName [DBN]** Specifies the name of a database to use when processing web requests, or uses the REQUIRED or AUTO keyword to specify whether database names are required as part of the URL.

  If this parameter is set to REQUIRED, the URL must specify the database name.

  If this parameter is set to AUTO, the URL may specify a database name, but does not need to do so. If the URL contains no database name, the default database on the server is used to process web requests.

  If this parameter is set to the name of a database, that database is used to process all web requests. The URL must not contain a database name.

- **LocalOnly [LOCAL]** When set to YES, this parameter causes a network database server to reject all connections from clients running on different computers. This option has no effect on personal database servers, which never accept web service requests from other computers. The default value is NO, which means accept requests from clients no matter where they are located.

- **LogFile [LOG]** The name of the file to which the database server is to write information about web service requests.

- **LogFormat [LF]** Controls the format of messages written to the log file and which fields appear in them. If they appear in the string, current values are substituted for the codes, such as @T, when each message is written.

  The default value is **@T - @W - @I - @P - "@M @U @V" - @R - @L - @E**, which produces messages such as the following:

  ```
  06/15 01:30:08.114 - 0.686 - 127.0.0.1 - 80
   - "GET /web/ShowTable HTTP/1.1" - 200 OK - 55133 -
  ```

  The format of the log file is compatible with Apache, so the same tools can be used to analyze it.

  For more information about field codes, see "LogFormat protocol option [LF]" [*SQL Anywhere Server - Database Administration*].

- **LogOptions [LOPT]** Allows you to specify keyword and error numbers that control which messages, or types of messages, are written to the log file.

  For more information, see "LogOptions protocol option [LOPT]" [*SQL Anywhere Server - Database Administration*].

For a complete list of the available options and detailed information about them, see "Network protocol options" [*SQL Anywhere Server - Database Administration*].

Understanding how URLs are interpreted

Universal Resource Locators, or URLs, identify documents, such as HTML pages, available from SOAP or HTTP web services. The URLs used in SQL Anywhere follow the patterns familiar to you from browsing the web. Users browsing through a database server need not be aware that their requests are not being handled by a traditional stand-alone web server.

Although standard in format, SQL Anywhere database servers interpret URLs differently than standard web servers. The options you specify when you start the database server also affect their interpretation.

The general syntax of the URL is as follows:

{ **http** | **https** }**://**[*user***:***password***@**]*host*[**:***port*][**/***dbn*]**/***service-name*[*path* | *?searchpart*]

The following is an example URL: http://localhost:80/demo/XMLtable.

User and password

When a web service requires authentication, the user name and password can be passed directly as part of the URL by separating them with a colon and prepending them to the host name, much like an email address.

Host and port

Like all standard HTTP requests, the start of the URL contains the host name or IP number and, optionally, a port number. The IP address or host name, and port, should be the one on which your server is listening. The IP address is the address of a network card in the computer running SQL Anywhere. The port number will be the port number you specified using the -xs option when you started the database server. If you did not specify a port number, the default port number for that type of service is used. For example, the server listens by default on port 80 for HTTP requests.

For more information, see “-xs server option” [*SQL Anywhere Server - Database Administration*].

Database name

The next token, between the slashes, is usually the name of a database. This database must be running on the server and must contain web services.

The default database is used if no database name appears in the URL and the database name was not specified using the DBN connection parameter to the -xs server option.

The database name can be omitted only if the database server is running only one database, or if the database name was specified using the DBN connection parameter to the -xs option.

Service name

The next portion of the URL is the service name. This service must exist in the specified database. The service name can extend beyond the next slash character because web service names can contain slash characters. SQL Anywhere matches the remainder of the URL with the defined services.

If the URL provides no service name, then the database server looks for a service named root. If the named service, or the root service, is not defined, then the server returns a `404 Not Found` error.

Parameters

Depending on the type of the target service, parameters can be supplied in different ways. Parameters to HTML, XML, and RAW services can be passed in any of the following ways:

- appended to the URL using slashes
- supplied as an explicit URL parameters list
- supplied as POST data in a POST request

Parameters to SOAP services must be included as part of a standard SOAP request. Values supplied in other ways are ignored.

URL path

To access parameter values, parameters must be given names. These host variable names, prefixed with a colon (:), can be included in the statement that forms part of the web service definition.

For example, suppose you define the following stored procedure:

```
CREATE PROCEDURE Display (IN ident INT )
BEGIN
    SELECT ID, GivenName, Surname FROM Customers
    WHERE ID = ident;
END;
```

A statement that calls the stored procedure requires a customer identification number. Define the service as follows:

```
CREATE SERVICE DisplayCustomer
TYPE 'HTML'
URL PATH ELEMENTS
AUTHORIZATION OFF
USER DBA
AS CALL Display( :url1 );
```

An example of a URL for this is: http://localhost/demo/DisplayCustomer/105.

The parameter `105` is passed as `url1` to the service. The clause URL PATH ELEMENTS indicates that parameters separated by slashes should be passed as parameters `url1`, `url2`, `url3`, and so on. Up to 10 parameters can be passed in this way.

Since there is only one parameter to the Display procedure, the service could have been defined like this:

```
CREATE SERVICE DisplayCustomer
TYPE 'HTML'
URL PATH ON
AUTHORIZATION OFF
USER DBA
AS CALL Display( :url );
```

In this case, the parameter `105` would be passed as `url` to the service. The clause URL PATH ON indicates that everything after the service name should be passed as a single parameter called `url`. So in the following URL, the string 105/106 would be passed as `url` (and a SQL error would result since the Display stored procedure requires an integer value).

http://localhost:80/demo/DisplayCustomer/105/106

For more information about variables, see "Working with variables" on page 782.

Parameters can also be accessed using the HTTP_VARIABLE function. For more information, see "HTTP_VARIABLE function [HTTP]" [*SQL Anywhere Server - SQL Reference*].

URL searchpart

Another method for passing parameters is through the URL searchpart mechanism. A URL searchpart consists of a question mark (?) followed by *name=value* pairs separated by ampersands (&). The searchpart is appended to the end of a URL. The following example shows the general format:

```
http://server/path/document?name1=value1&name2=value2
```

GET requests are formatted in this manner. If present, the named variables are defined and assigned the corresponding values.

For example, a statement that calls the stored procedure ShowSalesOrderDetail requires both a customer identification number and a product identification number:

```
CREATE SERVICE ShowSalesOrderDetail
TYPE 'HTML'
URL PATH OFF
AUTHORIZATION OFF
USER DBA
AS CALL ShowSalesOrderDetail( :customer_id, :product_id );
```

An example of a URL for this is: http://localhost:80/demo/ShowSalesOrderDetail?customer_id=101&product_id=300.

If you have URL PATH set to ON or ELEMENTS, additional variables are defined. However, the two are usually otherwise independent. You can allow variables to be used in requested URLs by setting URL PATH to ON or ELEMENTS. The following example illustrates how the two can be mixed:

```
CREATE SERVICE ShowSalesOrderDetail2
TYPE 'HTML'
URL PATH ON
AUTHORIZATION OFF
USER DBA
AS CALL ShowSalesOrderDetail( :customer_id, :url );
```

In the following example, both searchpart and URL path are used. The value 300 is assigned to `url` and 101 is assigned to `customer_id`.

http://localhost:80/demo/ShowSalesOrderDetail2/300?customer_id=101

This can also be expressed using searchpart only in the following manner.

http://localhost:80/demo/ShowSalesOrderDetail2/?customer_id=101&url=300

This then leads to the question of what happens when both are specified for the same variable. In the following example, first 300 and then 302 are assigned to `url` in sequence and it is the last assignment that takes precedence.

http://localhost:80/demo/ShowSalesOrderDetail2/300?customer_id=101&url=302

For more information about variables, see "Working with variables" on page 782.

Parameters can also be accessed using the HTTP_VARIABLE function. For more information, see "HTTP_VARIABLE function [HTTP]" [*SQL Anywhere Server - SQL Reference*].

Creating SOAP and DISH web services

SOAP and DISH web services are the means by which you create standard SOAP web services that can be accessed by standard SOAP clients, such as those written with Microsoft .NET or JAX-WS.

SOAP services

SOAP services are the mechanism for constructing web services in SQL Anywhere that accept and process standard SOAP requests.

To declare a SOAP service, specify that the service is to be of type SOAP. The body of a standard SOAP request is SOAP envelope, meaning an XML document with a specific format. SQL Anywhere parses and processes these requests using the procedures you provide. The response is automatically formatted in the form of a standard SOAP response, which is also a SOAP envelope, and returned to the client.

The syntax of the statement used to create SOAP services is as follows:

CREATE SERVICE *service-name*
TYPE 'SOAP'
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]
AS *statement*

DISH services

DISH services act as proxies for groups of SOAP services. In addition, they automatically construct WSDL (Web Services Description Language) documents for their clients that describe the SOAP services that they currently expose.

When you create a DISH service, the name given in the GROUP clause determines which SOAP services the DISH service exposes. Every SOAP service whose name is prefixed with the name of the DISH service is exposed. For example, specifying GROUP xyz exposes SOAP services xyz/aaaa, xyz/bbbb, or xyz/cccc. It does not expose SOAP services named abc/aaaa or xyzaaaa. SOAP services can be exposed by more than one DISH service. If no group name is specified, the DISH service exposes all the SOAP services in the database. The same characters are permitted in DISH group names as in SOAP service names.

The syntax of the statement used to create DISH services is as follows:

CREATE SERVICE *service-name*
TYPE 'DISH'
[**GROUP** { *group-name* | **NULL** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]

SOAP and DISH service formats

The FORMAT clause of the CREATE SERVICE statement customizes the SOAP service data payload to best suit the various types of SOAP clients, such as .NET and JAX-WS. The FORMAT clause affects the content of the WSDL document returned by a DISH service and the format of data payloads returned in SOAP responses.

The default format, DNET, is a native format for use with .NET SOAP client applications, which expect a .NET DataSet format.

The CONCRETE format is for use with clients such as JAX-WS and .NET that automatically generate interfaces based on the format of the returned data structures. When you specify this format, the WSDL document returned by SQL Anywhere exposes a SimpleDataset element that describes a result set in concrete terms. This element is a containment hierarchy of a rowset composed of an array of rows, each containing an array of column elements.

The XML format is for use with SOAP clients that accept the SOAP response as one large string, and use an XML parser to locate and extract the required elements and values. This format is generally the most portable between different types of SOAP clients.

If the format of a SOAP service is not specified, the format is inherited from the service's DISH service declaration. If the DISH service also does not declare a format, it defaults to DNET, which is compatible with .NET clients. A SOAP service that does not declare a format can be used with different types of SOAP clients by defining multiple DISH services, each having a different FORMAT type.

Creating homogeneous DISH services

SOAP services need not specify a format type—you can set the format type to NULL. In this case, the format is inherited from the DISH services that act as proxies for them. More than one DISH service can act as a proxy for each SOAP service, and these DISH services need not be of the same type. These facts mean that it is possible to use a single SOAP service with different types of SOAP clients, such as .NET and JAX-WS, by using multiple DISH services, each of a different type. Such DISH services are said to be **homogeneous** because they expose the same data payloads for the same SOAP services, but in different formats.

For example, consider the following two SOAP services, neither of which specifies a format:

```
CREATE SERVICE "abc/hello"
TYPE 'SOAP'
AS CALL hello(:student);

CREATE SERVICE "abc/goodbye"
TYPE 'SOAP'
AS CALL goodbye(:student);
```

Since neither of these services includes a FORMAT clause, the format, by default, is NULL. It is thus inherited from the DISH service that is acting as a proxy. Now, consider the following two DISH services:

```
CREATE SERVICE "abc_xml"
TYPE 'DISH'
GROUP "abc"
FORMAT 'XML';

CREATE SERVICE "abc_concrete"
TYPE 'DISH'
GROUP "abc"
FORMAT 'CONCRETE';
```

Since both DISH services specify the same group name abc, they act as proxies for the same SOAP services, namely all SOAP services whose names have the prefix "abc/".

However, when either of the two SOAP services is accessed through the abc_xml DISH service, the SOAP service inherits the XML format; when accessed through the abc_concrete SOAP service, the SOAP service inherits the CONCRETE format.

Homogeneous DISH services provide a means of avoiding duplicate services whenever you want to give different types of SOAP clients access to the SOAP web services you create.

Tutorial: Accessing web services from Microsoft .NET

The following tutorial demonstrates how to access web services from Microsoft .NET using Visual C#.

To create SOAP and DISH services

1. Copy the SQL Anywhere sample database from *samples-dir* to another location, such as *c:\webserver\demo.db*.
2. At a command prompt, execute the following statement to start a personal web server. The `-xs http(port=80)` option tells the database server to accept HTTP requests on port 80. If you already have a web server running on port 80, use another port number such as 8080 for this tutorial.

   ```
   dbeng11 -xs http(port=80) c:\webserver\demo.db
   ```
3. Start Interactive SQL. Connect to the SQL Anywhere sample database as the DBA. Execute the following statements:
   a. Define a SOAP service that lists the Employees table.

      ```
      CREATE SERVICE "SASoapTest/EmployeeList"
      TYPE 'SOAP'
      AUTHORIZATION OFF
      SECURE OFF
      USER DBA
      AS SELECT * FROM Employees;
      ```

      Because authorization has been turned off, anyone can use this service without supplying a user name and password. The commands run under user DBA. This arrangement is simple, but insecure.

   b. Create a DISH service to act as a proxy for the SOAP service and to generate the WSDL document.

      ```
      CREATE SERVICE "SASoapTest_DNET"
      TYPE 'DISH'
      GROUP "SASoapTest"
      FORMAT 'DNET'
      AUTHORIZATION OFF
      SECURE OFF
      USER DBA;
      ```

      The SOAP and DISH service must be of format DNET. In this example, the FORMAT clause was omitted when the SOAP service was created. As a result, the SOAP service inherits the DNET format from the DISH service.
4. Start Microsoft Visual Studio. Note that this example uses functions from the .NET Framework 2.0.
   a. Create a new Windows application project using Visual C#.

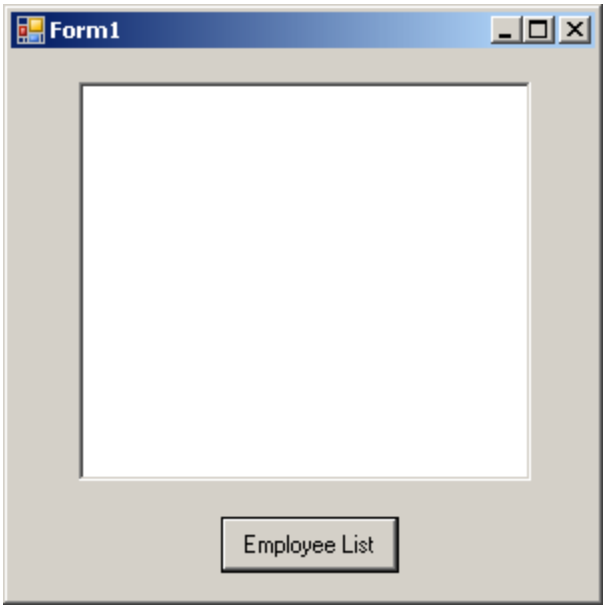
      An empty form appears.
   b. From the **Project** menu, choose **Add Web Reference**.
   c. In the **URL** field of the **Add Web Reference** page, enter the following URL: **http://localhost:80/demo/SASoapTest_DNET**.
   d. Click **Go**.

You are presented with a list of the methods available for SASoapTest_DNET. You should see the EmployeeList method.

e. Click **Add Reference** to finish.

The **Solution Explorer** window shows the new web reference.

f. Add a **ListBox** and a **Button** to the form as shown in the following diagram.



g. Rename the button text to **Employee List**.

h. Double-click the Employee List button and add the following code for the button click event.

```
int sqlCode;

listBox1.Items.Clear();

localhost.SASoapTest_DNET proxy = new localhost.SASoapTest_DNET();

DataSet results = proxy.EmployeeList(out sqlCode);
DataTableReader dr = results.CreateDataReader();
while (dr.Read())
{
  for (int i = 0; i < dr.FieldCount; i++)
  {
    string columnName = dr.GetName(i);
    try
    {
      string value = dr.GetString(i);
      listBox1.Items.Add(columnName + "=" + value);
    }
    catch ( InvalidCastException )
    {
      listBox1.Items.Add(columnName + "=(null)");
    }
  }
  listBox1.Items.Add("");
```



```
    }
    dr.Close();
```

i. Build and run the program.

   The listbox will display the EmployeeList result set as column name=value pairs.

   A try/catch block is included to handle NULL column values such as those found in the TerminationDate column of the Employees table.

Tutorial: Accessing web services from JAX-WS

The following tutorial demonstrates how to access web services using the Java API for XML Web Services (JAX-WS).

Before you begin, you will need the JAX-WS tools available from Sun. If you do not have this package installed on your system, you must download the JAX-WS tools and install them. To download the JAX-WS tools, visit http://java.sun.com/webservices/. Click the link for JAX-WS. This should take you to the Java API for XML Web Services page. Click the Download Now link. After you have downloaded the software package, install it on your system.

This example was developed with JAX-WS 2.1.3 for Windows.

SQL Anywhere SOAP web services that are accessed from JAX-WS should be declared to be of format CONCRETE.

To create SOAP and DISH services

1. At a command prompt, execute the following statement to start a personal web server. The `-xs http(port=80)` option tells the database server to accept HTTP requests on port 80. If you already have a web server running on port 80, use another port number such as 8080 for this tutorial.

   ```
   dbeng11 -xs http(port=80) samples-dir\demo.db
   ```

2. Start Interactive SQL and connect to the SQL Anywhere sample database as the DBA. Execute the following statements:

   a. Define a stored procedure that lists some columns of the Employees table.

   ```
   CREATE PROCEDURE ListEmployees()
   RESULT (
    EmployeeID            INTEGER,
    Surname               CHAR(20),
    GivenName             CHAR(20),
    StartDate             DATE,
    TerminationDate       DATE )
   BEGIN
     SELECT EmployeeID, Surname, GivenName,
       StartDate, TerminationDate
     FROM Employees;
   END;
   ```

   b. Define a SOAP service that calls this stored procedure.

   ```
   CREATE SERVICE "WS/EmployeeList"
   TYPE 'SOAP'
   FORMAT 'CONCRETE' EXPLICIT OFF
   DATATYPE OUT
   AUTHORIZATION OFF
   SECURE OFF
   USER DBA
   AS CALL ListEmployees();
   ```

   The EXPLICIT clause can only be used with a SOAP or DISH service of type CONCRETE. In this example, EXPLICIT OFF indicates that the corresponding DISH service should generate XML Schema that describes the generic SimpleDataset object. This option only affects the WSDL document that is generated. Later on, we will look at an example that uses EXPLICIT ON. See “Tutorial: Using data types with JAX-WS” on page 757.

DATATYPE OUT indicates that explicit data type information is generated in the XML result set response. If DATATYPE OFF had been specified, then all data would be typed as string. This option does not affect the WSDL document that is generated.

Because authorization has been turned off, anyone can use this service without supplying a user name and password. The commands run under user DBA. This arrangement is simple, but not secure.

c. Create a DISH service to act as a proxy for the SOAP service and to generate the WSDL document.

```
CREATE SERVICE "WSDish"
TYPE 'DISH'
FORMAT 'CONCRETE'
GROUP "WS"
AUTHORIZATION OFF
SECURE OFF
USER DBA;
```

The SOAP and DISH service must both be format CONCRETE. Since the EmployeeList service is in the WS group, a GROUP clause is included.

3. Take a look at the WSDL that the DISH service automatically creates. To do so, open a web browser and browse to the following URL: http://localhost:80/demo/WSDish. The DISH service automatically generates a WSDL document that appears in the browser window.

   In particular, observe the SimpleDataset object that is exposed because the format of this service is CONCRETE and EXPLICIT is OFF. In a later step, the **wsimport** application uses this information to generate a SOAP 1.1 client interface for this service.

```
<s:complexType name="SimpleDataset">
<s:sequence>
<s:element name="rowset">
  <s:complexType>
  <s:sequence>
  <s:element name="row" minOccurs="0" maxOccurs="unbounded">
    <s:complexType>
    <s:sequence>
      <s:any minOccurs="0" maxOccurs="unbounded" />
    </s:sequence>
    </s:complexType>
  </s:element>
  </s:sequence>
  </s:complexType>
</s:element>
</s:sequence>
</s:complexType>
```

To generate a JAX-WS interface for these web services

1. In this example, the Java API for XML Web Services (JAX-WS) and the Sun Java 1.6.0 JDK are installed on drive *C:*. Set your PATH environment variable so that it includes the JAX-WS binaries, as well as the JDK binaries. The binaries are located in the following directories:

```
c:\Sun\jaxws-ri\bin
c:\jdk1.6.0\bin
```

2. Set your CLASSPATH environment variable.

```
SET classpath=.;C:\Sun\jaxws-ri\lib\jaxb-api.jar
;C:\Sun\jaxws-ri\lib\jaxws-rt.jar
```

3. The next step is to generate the interface needed to call the web service.

   At a command prompt, execute the following command.

   ```
   wsimport -keep -Xendorsed "http://localhost:80/demo/WSDish"
   ```

   The wsimport tool retrieves the WSDL document from the given URL, generates the Java files that define the interface for it, and then compiles the Java files.

   The **keep** option tells wsimport not to delete the *.java* files. Without this option, these files are deleted after the corresponding *.class* files have been generated. Saving these files makes it easer to examine the makeup of the interface.

   The **Xendorsed** option allows you to use the JAX-WS 2.1 API with JDK6.

   Once this command completes, you should have a new subdirectory structure named *localhost\demo \ws* in your current directory that contains the following Java files, along with the compiled *.class* versions of each Java file.

   ```
   EmployeeList.java
   EmployeeListResponse.java
   FaultMessage.java
   ObjectFactory.java
   package-info.java
   SimpleDataset.java
   WSDish.java
   WSDishSoapPort.java
   ```

To use the generated JAX-WS interface

1. Save the following Java source code into *SASoapDemo.java*. Make sure that this file is located in the same directory containing the *localhost* subdirectory that was generated by the wsimport tool.

   ```
   // SASoapDemo.java illustrates a web service client that
   // calls the WSDish service and prints out the data.

   import java.util.*;
   import javax.xml.ws.*;
   import org.w3c.dom.Element;
   import org.w3c.dom.Node;
   import localhost.demo.ws.*;

   public class SASoapDemo
   {
     @WebServiceRef( wsdlLocation= "http://localhost:8080/demo/WSDish" )
     public static void main( String[] args )
     {
       try {
         WSDish service = new WSDish();

         Holder<SimpleDataset> response = new Holder<SimpleDataset>();
         Holder<Integer> sqlcode = new Holder<Integer>();

         WSDishSoapPort port = service.getWSDishSoap();

         // This is the SOAP service call to EmployeeList
         port.employeeList( response, sqlcode );
   ```



```
        localhost.demo.ws.SimpleDataset result = response.value;
        SimpleDataset.Rowset rowset = result.getRowset();

        List<SimpleDataset.Rowset.Row> rows = rowset.getRow();

        for ( int i = 0; i < rows.size(); i++ ) {
          SimpleDataset.Rowset.Row row = rows.get( i );
          List<Object> cols = row.getAny();

          System.out.println( "Number of columns=" + cols.size() );

          for ( int j = 0; j < cols.size(); j++ ) {
            // Column data is contained as a SOAPElement
            Element col = (Element)cols.get(j);
            System.out.print(col.getLocalName() + "=" );
            Node node = col.getFirstChild();
            if( node == null ) {
              System.out.println( "(null)" );
            } else {
              System.out.println( node.getNodeValue() );
            }
          }
          System.out.println();
        }

      }
      catch (Exception x) {
        x.printStackTrace();
      }
    }
  }
```

If you chose to start the web server using a different port number such as 8080, then you will have to alter the `import localhost` source line to something like the following:

```
import localhost._8080.demo.ws.*;
```

2. Compile *SASoapDemo.java*.

   ```
   javac SASoapDemo.java
   ```

3. Run the compiled class file.

   ```
   java SASoapDemo
   ```

   When the application sends its request to the web server, it receives an XML result set response that consists of an EmployeeListResult with a rowset containing several row entries. Also included in the response is the sqlcode result from executing the query. An example of the response is shown next.

```
<tns:EmployeeListResponse>
 <tns:EmployeeListResult xsi:type='tns:SimpleDataset'>
  <tns:rowset>
    <tns:row>...</tns:row>
.
.
.
    <tns:row>...</tns:row>
   </tns:rowset>
 </tns:EmployeeListResult>
 <tns:sqlcode>0</tns:sqlcode>
</tns:EmployeeListResponse>
```

Each row in the rowset is sent in a format similar to the following.

```
<tns:row>
   <tns:EmployeeID xsi:type="xsd:int">1751</tns:EmployeeID>
   <tns:Surname xsi:type="xsd:string">Ahmed</tns:Surname>
   <tns:GivenName xsi:type="xsd:string">Alex</tns:GivenName>
   <tns:StartDate xsi:type="xsd:date">1994-07-12-04:00</tns:StartDate>
   <tns:TerminationDate xsi:type="xsd:date">2008-04-18-04:00
       </tns:TerminationDate>
</tns:row>
```

Note that the column name and data type are included.

Using a proxy
You can observe the response shown above through the use of proxy software that logs the XML message traffic. The proxy inserts itself between your client application and the web server.

The EmployeeList result set is displayed as (type)column name=value pairs by the SASoapDemo application. Several lines of output similar to the following should be generated.

The EmployeeList result set is displayed as column name=value pairs. Several lines of output similar to the following should be generated.

```
Number of columns=4
EmployeeID=102
Surname=Whitney
GivenName=Fran
StartDate=1984-08-28-04:00

Number of columns=4
EmployeeID=105
Surname=Cobb
GivenName=Matthew
StartDate=1985-01-01-05:00
.
.
.
Number of columns=4
EmployeeID=1740
Surname=Nielsen
GivenName=Robert
StartDate=1994-06-24-04:00

Number of columns=5
EmployeeID=1751
Surname=Ahmed
GivenName=Alex
StartDate=1994-07-12-04:00
TerminationDate=2008-04-18-04:00
```

Note that the TerminationDate column is only sent when its value is not NULL. For this example, the last row in the Employees table was altered such that a non-NULL termination date was set.

Also note that date values include an offset from UTC time. In the above sample data, the server is located in the North American Eastern time zone. This is 5 hours earlier than UTC time (-05:00) for dates when daylight savings is not in effect and 4 hours earlier than UTC time (-04:00) for dates when daylight savings is in effect.

Using procedures that provide HTML documents

Generally, it is easiest to write a procedure that handles the requests sent to a particular service. Such a procedure should return a web page. Optionally the procedure can accept arguments, passed as part of the URL, to customize its output.

The following example, however, is much simpler. It demonstrates how simple a service can be. This web service simply returns the phrase "Hello world!".

```
CREATE SERVICE hello
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS SELECT 'Hello world!';
```

Start a database server with the -xs option to enable handling of web requests, and then request the URL http://localhost/hello from any web browser. The words Hello world! appear on an otherwise plain page.

HTML pages

The above page appears in your browser in plain text. This happens because the default HTTP Content-Type is text/plain. To create a more normal web page, formatted in HTML, you must do two things:

- Set the HTTP Content-Type header field to text/html so that the browsers expect HTML.
- Include HTML tags in the output.

You can write tags to the output in two ways. One way is to use the phrase TYPE 'HTML' in the CREATE SERVICE statement. Doing so instructs the SQL Anywhere database server to add HTML tags for you. This can work quite well if, for example, you are returning a table.

The other way is to use TYPE 'RAW' and write out all the necessary tags yourself. This second method provides the most control over the output. Note that specifying type RAW does not necessarily mean the output is not in HTML or XML format. It only tells SQL Anywhere that it can pass the return value directly to the client without adding tags itself.

The following procedure generates a fancier version of Hello world. For convenience, the body of the work is done in the following procedure, which formats the web page.

The built-in procedure sa_set_http_header is used to set the HTTP header type so browsers interpret the result correctly. If you omit this statement, your browser displays all the HTML codes, rather than using them to format the document.

```
CREATE PROCEDURE hello_pretty_world ()
RESULT (html_doc XML)
BEGIN
  CALL dbo.sa_set_http_header( 'Content-Type', 'text/html' );
  SELECT HTML_DECODE(
    XMLCONCAT(
      '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">',
      XMLELEMENT('HTML',
      XMLELEMENT('HEAD',
          XMLELEMENT('TITLE', 'Hello Pretty World')
        ),
        XMLELEMENT('BODY',
          XMLELEMENT('H1', 'Hello Pretty World!'),
```

```
            XMLELEMENT('P',
              '(If you see the tags in your browser, check that '
              || 'the Content-Type header is set to text/html.)'
            )
          )
        )
      )
    );
END
```

The following statement creates a service that uses this procedure. The statement calls the above procedure, which generates the Hello Pretty World web page.

```
CREATE SERVICE hello_pretty_world
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS CALL hello_pretty_world();
```

Once you have created the procedure and the service, you are ready to access the web page. Ensure that your database server was started with the correct -xs option values, and then open the URL http://localhost/hello_pretty_world in a web browser.

You see the results formatted in a simple HTML page, with the title Hello Pretty World. You can make the web page as elaborate as you want by including more content, using more tags, using style sheets, or including scripts that run in the browser. In all cases, you create the necessary services to handle the browser's requests.

For more information about built-in stored procedures, see "Alphabetical list of system procedures" [*SQL Anywhere Server - SQL Reference*].

Root services

When no service name is included in a URL, SQL Anywhere looks for a web service named **root**. The role of root pages is analogous to the role of *index.html* pages in many traditional web servers.

Root services are handy for creating home pages because they can handle URL requests that contain only the address of your web site. For example, the following procedure and service implement a simple web page that appears when you browse to the URL http://localhost.

```
CREATE PROCEDURE HomePage()
RESULT (html_doc XML)
BEGIN
  CALL dbo.sa_set_http_header( 'Content-Type', 'text/html' );
  SELECT HTML_DECODE(
    XMLCONCAT(
      '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">',
      XMLELEMENT('HTML',
        XMLELEMENT('HEAD',
          XMLELEMENT('TITLE', 'My Home Page')
        ),
        XMLELEMENT('BODY',
          XMLELEMENT('H1', 'My home on the web'),
          XMLELEMENT('P',
            'Thank you for visiting my web site!'
          )
        )
      )
    )
```



```
  );
END
```

Now create a service that uses this procedure:

```
CREATE SERVICE root
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS CALL HomePage();
```

You can access this web page by browsing to the URL http://localhost, as long as you do not specify that database names are mandatory when you start the database server.

For more information, see "Starting a database server that listens for web requests" on page 725.

Examples

More extensive examples are included in the *samples-dir\SQLAnywhere\HTTP* directory.

Working with data types

By default, the XML encoding of parameter input is string and the result set output for SOAP service formats contains no information that specifically describes the data type of the columns in the result set. For all formats, parameter data types are string. For the DNET format, within the schema section of the response, all columns are typed as string. CONCRETE and XML formats contain no data type information in the response. This default behavior can be manipulated using the DATATYPE clause.

SQL Anywhere enables data typing using the DATATYPE clause. Data type information can be included in the XML encoding of parameter input and result set output or responses for all SOAP service formats. This simplifies parameter passing from SOAP toolkits by not requiring client code to explicitly convert parameters to Strings. For example, an integer can be passed as an int. XML encoded data types enable a SOAP toolkit to parse and cast the data to the appropriate type.

When using string data types exclusively, the application needs to implicitly know the data type for every column within the result set. This is not necessary when data typing is requested of the web server. To control whether data type information is included, the DATATYPE clause can be used when the web service is defined.

DATATYPE { **OFF** | **ON** | **IN** | **OUT** }

- **OFF** For DNET output format, SQL Anywhere data types are translated to and from XML Schema string types. For CONCRETE and XML formats, no data type information is emitted.
- **ON** This is the default behavior when the DATATYPE option is not used. Data type information is emitted for both input parameters and result set responses. SQL Anywhere data types are translated to and from XML Schema data types.
- **IN** Data type information is emitted for input parameters only.
- **OUT** Data type information is emitted for result set responses only.

Here is an example of a web service definition that enlists data typing for the result set response.

```
CREATE SERVICE "SASoapTest/EmployeeList"
TYPE 'SOAP'
AUTHORIZATION OFF
SECURE OFF
USER DBA
DATATYPE OUT
AS SELECT * FROM Employees;
```

In this example, data type information is requested for result set responses only since this service does not have parameters.

Data typing is applicable to all SQL Anywhere web services defined as type 'SOAP'.

Data typing of input parameters

Data typing of input parameters is supported by simply exposing the parameter data types as their true data types in the WSDL generated by the DISH service.

A typical string parameter definition (or a non-typed parameter) would look like the following:


```
<s:element minOccurs="0" maxOccurs="1" name="a_varchar" nillable="true"
type="s:string" />
```

The String parameter may be nillable, that is, it may or may not occur.

For a typed parameter such as an integer, the parameter must occur and is not nillable. The following is an example.

```
<s:element minOccurs="1" maxOccurs="1" name="an_int" nillable="false"
type="s:int" />
```

Data typing of output parameters

All SQL Anywhere web services of type 'SOAP' may expose data type information within the response data. The data types are exposed as attributes within the rowset column element.

The following is an example of a typed SimpleDataSet response from a SOAP FORMAT 'CONCRETE' web service.

```
<SOAP-ENV:Body>
  <tns:test_types_concrete_onResponse>
   <tns:test_types_concrete_onResult xsi:type='tns:SimpleDataset'>
    <tns:rowset>
      <tns:row>
       <tns:lvc xsi:type="xsd:string">Hello World</tns:lvc>
       <tns:i xsi:type="xsd:int">99</tns:i>
       <tns:ii xsi:type="xsd:long">99999999</tns:ii>
       <tns:f xsi:type="xsd:float">3.25</tns:f>
       <tns:d xsi:type="xsd:double">.555555555555555582</tns:d>
       <tns:bin xsi:type="xsd:base64Binary">AAAAZg==</tns:bin>
       <tns:date xsi:type="xsd:date">2006-05-29-04:00</tns:date>
      </tns:row>
     </tns:rowset>
   </tns:test_types_concrete_onResult>
  <tns:sqlcode>0</tns:sqlcode>
  </tns:test_types_concrete_onResponse>
</SOAP-ENV:Body>
```

The following is an example of a response from a SOAP FORMAT 'XML' web service returning the XML data as a string. The interior rowset consists of encoded XML and is presented here in its decoded form for legibility.

```
<SOAP-ENV:Body>
 <tns:test_types_XML_onResponse>
  <tns:test_types_XML_onResult xsi:type='xsd:string'>
   <tns:rowset
     xmlns:tns="http://localhost/satest/dish"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <tns:row>
     <tns:lvc xsi:type="xsd:string">Hello World</tns:lvc>
     <tns:i xsi:type="xsd:int">99</tns:i>
     <tns:ii xsi:type="xsd:long">99999999</tns:ii>
     <tns:f xsi:type="xsd:float">3.25</tns:f>
     <tns:d xsi:type="xsd:double">.555555555555555582</tns:d>
     <tns:bin xsi:type="xsd:base64Binary">AAAAZg==</tns:bin>
     <tns:date xsi:type="xsd:date">2006-05-29-04:00</tns:date>
    </tns:row>
   </tns:rowset>
  </tns:test_types_XML_onResult>
  <tns:sqlcode>0</tns:sqlcode>
```

```
 </tns:test_types_XML_onResponse>
</SOAP-ENV:Body>
```

Note that, in addition to the data type information, the namespace for the elements and the XML schema provides all the information necessary for post processing by an XML parser. When no data type information exists in the result set (DATATYPE OFF or IN) then the xsi:type and the XML schema namespace declarations are omitted.

An example of a SOAP FORMAT 'DNET' web service returning a typed SimpleDataSet follows:

```
<SOAP-ENV:Body>
  <tns:test_types_dnet_outResponse>
   <tns:test_types_dnet_outResult xsi:type='sqlresultstream:SqlRowSet'>
    <xsd:schema id='Schema2'
       xmlns:xsd='http://www.w3.org/2001/XMLSchema'
       xmlns:msdata='urn:schemas-microsoft.com:xml-msdata'>
     <xsd:element name='rowset' msdata:IsDataSet='true'>
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name='row' minOccurs='0' maxOccurs='unbounded'>
         <xsd:complexType>
          <xsd:sequence>
           <xsd:element name='lvc' minOccurs='0' type='xsd:string' />
           <xsd:element name='ub' minOccurs='0' type='xsd:unsignedByte' />
           <xsd:element name='s' minOccurs='0' type='xsd:short' />
           <xsd:element name='us' minOccurs='0' type='xsd:unsignedShort' />
           <xsd:element name='i' minOccurs='0' type='xsd:int' />
           <xsd:element name='ui' minOccurs='0' type='xsd:unsignedInt' />
           <xsd:element name='l' minOccurs='0' type='xsd:long' />
           <xsd:element name='ul' minOccurs='0' type='xsd:unsignedLong' />
           <xsd:element name='f' minOccurs='0' type='xsd:float' />
           <xsd:element name='d' minOccurs='0' type='xsd:double' />
           <xsd:element name='bin' minOccurs='0' type='xsd:base64Binary' />
           <xsd:element name='bool' minOccurs='0' type='xsd:boolean' />
           <xsd:element name='num' minOccurs='0' type='xsd:decimal' />
           <xsd:element name='dc' minOccurs='0' type='xsd:decimal' />
           <xsd:element name='date' minOccurs='0' type='xsd:date' />
          </xsd:sequence>
         </xsd:complexType>
        </xsd:element>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
    </xsd:schema>
    <diffgr:diffgram xmlns:msdata='urn:schemas-microsoft-com:xml-msdata'
xmlns:diffgr='urn:schemas-microsoft-com:xml-diffgram-v1'>
     <rowset>
      <row>
       <lvc>Hello World</lvc>
       <ub>128</ub>
       <s>-99</s>
       <us>33000</us>
       <i>-2147483640</i>
       <ui>4294967295</ui>
       <l>-9223372036854775807</l>
       <ul>18446744073709551615</ul>
       <f>3.25</f>
       <d>.555555555555555582</d>
       <bin>QUJD</bin>
       <bool>1</bool>
       <num>123456.123457</num>
       <dc>-1.756000</dc>
```



```
        <date>2006-05-29-04:00</date>
       </row>
      </rowset>
     </diffgr:diffgram>
    </tns:test_types_dnet_outResult>
   <tns:sqlcode>0</tns:sqlcode>
   </tns:test_types_dnet_outResponse>
</SOAP-ENV:Body>
```

Mapping SQL Anywhere types to XML Schema types

SQL Anywhere type	XML Schema type	XML Example
CHAR	string	Hello World
VARCHAR	string	Hello World
LONG VARCHAR	string	Hello World
TEXT	string	Hello World
NCHAR	string	Hello World
NVARCHAR	string	Hello World
LONG NVARCHAR	string	Hello World
NTEXT	string	Hello World
XML	This is user defined. A parameter is assumed to be valid XML representing a complex type (for example, base64Binary, SOAP array, struct).	<inputHexBinary xsi:type="xsd:hexBinary"> 414243 </inputHexBinary> (interpreted as 'ABC')
UNIQUEIDENTIFIERSTR	string	12345678-1234-5678-9012-123456789012
BIGINT	long	-9223372036854775807
UNSIGNED BIGINT	unsignedLong	18446744073709551615
BIT	boolean	1
DECIMAL	decimal	-1.756000
DOUBLE	double	.555555555555555582
FLOAT	float	12.3456792831420898
INTEGER	int	-2147483640

SQL Anywhere type	XML Schema type	XML Example
UNSIGNED INTEGER	unsignedInt	4294967295
NUMERIC	decimal	123456.123457
REAL	float	3.25
SMALLINT	short	-99
UNSIGNED SMALLINT	unsignedShort	33000
TINYINT	unsignedByte	128
MONEY	decimal	12345678.9900
SMALLMONEY	decimal	12.3400
VARBIT	string	11111111
LONG VARBIT	string	00000000000000001000000000000000
DATE	date	2006-11-21-05:00
DATETIME	dateTime	2006-05-21T09:00:00.000-08:00
SMALLDATETIME	dateTime	2007-01-15T09:00:00.000-08:00
TIME	time	14:14:48.980-05:00
TIMESTAMP	dateTime	2007-01-12T21:02:14.420-06:00
BINARY	base64Binary	AAAAZg==
IMAGE	base64Binary	AAAAZg==
LONG BINARY	base64Binary	AAAAZg==
UNIQUEIDENTIFIER	string	12345678-1234-5678-9012-123456789012
VARBINARY	base64Binary	AAAAZg==

Mapping XML Schema types to Java types

XML Schema Type	Java Data Type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger

XML Schema Type	Java Data Type
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	javax.xml.datatype.XMLGregorianCalendar
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:g	javax.xml.datatype.XMLGregorianCalendar
xsd:anySimpleType	java.lang.Object
xsd:anySimpleType	java.lang.String
xsd:duration	javax.xml.datatype.Duration
xsd:NOTATION	javax.xml.namespace.QName

Tutorial: Using data types with Microsoft .NET

The following tutorial demonstrates how to use the SQL Anywhere web service datatype support from within Microsoft .NET using Visual C#.

To create SOAP and DISH services

1. Copy the SQL Anywhere sample database from *samples-dir* to another location, such as *c:\webserver\demo.db*.
2. At a command prompt, execute the following statement to start a personal web server. The `-xs http(port=80)` option tells the database server to accept HTTP requests on port 80. If you already have a web server running on port 80, use another port number such as 8080 for this tutorial.

   ```
   dbeng11 -xs http(port=80) c:\webserver\demo.db
   ```
3. Start Interactive SQL. Connect to the SQL Anywhere sample database as the DBA. Execute the following statements:
   a. Define a SOAP service that lists the Employees table.

   ```
   CREATE SERVICE "SASoapTest/EmployeeList"
   TYPE 'SOAP'
   AUTHORIZATION OFF
   SECURE OFF
   USER DBA
   DATATYPE OUT
   AS SELECT * FROM Employees;
   ```

   In this example, DATATYPE OUT is specified to generate datatype information in the XML result set response. A fragment of the response from the web server is shown below. Note that the type information matches the data type of the database columns.

   ```
   <xsd:element name='EmployeeID' minOccurs='0' type='xsd:int' />
   <xsd:element name='ManagerID' minOccurs='0' type='xsd:int' />
   <xsd:element name='Surname' minOccurs='0' type='xsd:string' />
   <xsd:element name='GivenName' minOccurs='0' type='xsd:string' />
   <xsd:element name='DepartmentID' minOccurs='0' type='xsd:int' />
   <xsd:element name='Street' minOccurs='0' type='xsd:string' />
   <xsd:element name='City' minOccurs='0' type='xsd:string' />
   <xsd:element name='State' minOccurs='0' type='xsd:string' />
   <xsd:element name='Country' minOccurs='0' type='xsd:string' />
   <xsd:element name='PostalCode' minOccurs='0' type='xsd:string' />
   <xsd:element name='Phone' minOccurs='0' type='xsd:string' />
   <xsd:element name='Status' minOccurs='0' type='xsd:string' />
   <xsd:element name='SocialSecurityNumber' minOccurs='0'
   type='xsd:string' />
   <xsd:element name='Salary' minOccurs='0' type='xsd:decimal' />
   <xsd:element name='StartDate' minOccurs='0' type='xsd:date' />
   <xsd:element name='TerminationDate' minOccurs='0' type='xsd:date' />
   <xsd:element name='BirthDate' minOccurs='0' type='xsd:date' />
   <xsd:element name='BenefitHealthInsurance' minOccurs='0'
   type='xsd:boolean' />
   <xsd:element name='BenefitLifeInsurance' minOccurs='0'
   type='xsd:boolean' />
   <xsd:element name='BenefitDayCare' minOccurs='0' type='xsd:boolean' />
   <xsd:element name='Sex' minOccurs='0' type='xsd:string' />
   ```

   b. Create a DISH service to act as a proxy for the SOAP service and to generate the WSDL document.


```
CREATE SERVICE "SASoapTest_DNET"
TYPE 'DISH'
GROUP "SASoapTest"
FORMAT 'DNET'
AUTHORIZATION OFF
SECURE OFF
USER DBA;
```

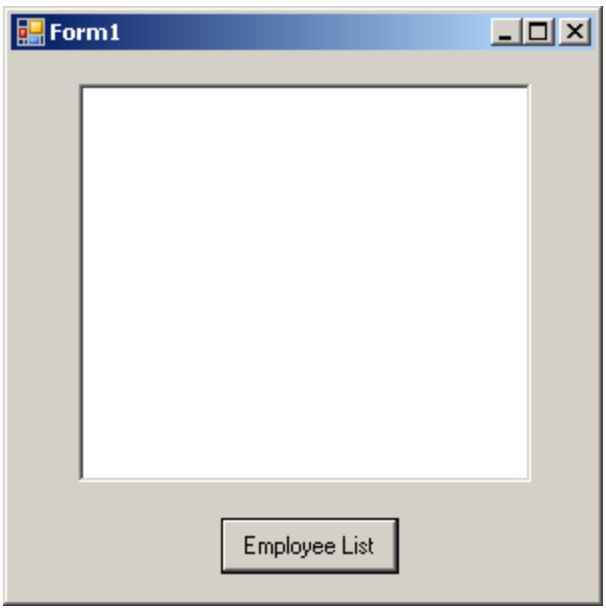
The SOAP and DISH service must be of format DNET. In this example, the FORMAT clause was omitted when the SOAP service was created. As a result, the SOAP service inherits the DNET format from the DISH service.

4. Start Microsoft Visual Studio. Note that this example uses functions from the .NET Framework 2.0.
   a. Create a new Windows Application project using Visual C#.

      An empty form appears.
   b. From the **Project** menu, choose **Add Web Reference**.
   c. In the **URL** field of the **Add Web Reference** page, enter the following URL: **http://localhost:80/demo/SASoapTest_DNET**.
   d. Click **Go**.

      You are presented with a list of the methods available for SASoapTest_DNET. You should see the EmployeeList method.
   e. Click **Add Reference** to finish.

      The **Solution Explorer** window shows the new web reference.
   f. Add a **ListBox** and a **Button** to the form as shown in the following diagram.



   g. Rename the button text to **Employee List**.
   h. Double-click the Employee List button and add the following code for the button click event.

```
int sqlCode;

listBox1.Items.Clear();

localhost.SASoapTest_DNET proxy = new localhost.SASoapTest_DNET();

DataSet results = proxy.EmployeeList(out sqlCode);
DataTableReader dr = results.CreateDataReader();
while (dr.Read())
{
  for (int i = 0; i < dr.FieldCount; i++)
  {
    string columnName = "(" + dr.GetDataTypeName(i)
                      + ")"
                      + dr.GetName(i);
    if (dr.IsDBNull(i))
    {
      listBox1.Items.Add(columnName + "=(null)");
    }
    else {
      System.TypeCode typeCode =
          System.Type.GetTypeCode(dr.GetFieldType(i));
      switch (typeCode)
      {
        case System.TypeCode.Int32:
          Int32 intValue = dr.GetInt32(i);
          listBox1.Items.Add(columnName + "="
              + intValue);
          break;
        case System.TypeCode.Decimal:
          Decimal decValue = dr.GetDecimal(i);
          listBox1.Items.Add(columnName + "="
              + decValue.ToString("c"));
          break;
        case System.TypeCode.String:
          string stringValue = dr.GetString(i);
          listBox1.Items.Add(columnName + "="
              + stringValue);
          break;
        case System.TypeCode.DateTime:
          DateTime dateValue = dr.GetDateTime(i);
          listBox1.Items.Add(columnName + "="
              + dateValue);
          break;
        case System.TypeCode.Boolean:
          Boolean boolValue = dr.GetBoolean(i);
          listBox1.Items.Add(columnName + "="
              + boolValue);
          break;
        case System.TypeCode.DBNull:
          listBox1.Items.Add(columnName
              + "=(null)");
          break;
        default:
          listBox1.Items.Add(columnName
              + "=(unsupported)");
          break;
      }
    }
  }
  listBox1.Items.Add("");
}
dr.Close();
```


This example is designed to illustrate the fine control the application developer can have over the datatype information that is available.

i. Build and run the program.

The XML response from the web server includes a formatted result set. The first row of the formatted result set is shown below.

```
<row>
 <EmployeeID>102</EmployeeID>
 <ManagerID>501</ManagerID>
 <Surname>Whitney</Surname>
 <GivenName>Fran</GivenName>
 <DepartmentID>100</DepartmentID>
 <Street>9 East Washington Street</Street>
 <City>Cornwall</City>
 <State>NY</State>
 <Country>USA</Country>
 <PostalCode>02192</PostalCode>
 <Phone>6175553985</Phone>
 <Status>A</Status>
 <SocialSecurityNumber>017349033</SocialSecurityNumber>
 <Salary>45700.000</Salary>
 <StartDate>1984-08-28-05:00</StartDate>
 <TerminationDate xsi:nil="true" />
 <BirthDate>1958-06-05-05:00</BirthDate>
 <BenefitHealthInsurance>1</BenefitHealthInsurance>
 <BenefitLifeInsurance>1</BenefitLifeInsurance>
 <BenefitDayCare>0</BenefitDayCare>
 <Sex>F</Sex>
</row>
```

There are a few things to note about the XML result set response.

- All column data is converted to a string representation of the data.
- Columns that contain date and/or time information include the offset from UTC of the web server. In this example, the offset is -05:00 which is 5 hours to the west of UTC (in this case, North American Eastern Standard Time).
- Columns containing date only are formatted as follows:`yyyy-mm-dd-HH:MM` or `yyyy-mm-dd+HH:MM`. A zone offset (-HH:MM or +HH:MM) is suffixed to the string.
- Columns containing time only are formatted as follows:`hh:mm:ss.nnn-HH:MM` or `hh:mm:ss.nnn+HH:MM`. A zone offset (-HH:MM or +HH:MM) is suffixed to the string.
- Columns containing date and time are formatted as follows:`yyyy-mm-ddThh:mm:ss.nnn-HH:MM` or `yyyy-mm-ddThh:mm:ss.nnn+HH:MM`. Note that the date is separated from the time using the letter 'T'. A zone offset (-HH:MM or +HH:MM) is suffixed to the string.

The listbox will display the EmployeeList result set as (type)column name=value pairs. The result from processing the first row of the result set is shown below.

```
(Int32)EmployeeID=102
(Int32)ManagerID=501
(String)Surname=Whitney
(String)GivenName=Fran
(Int32)DepartmentID=100
(String)Street=9 East Washington Street
(String)City=Cornwall
(String)State=New York
(String)Country=USA
```

```
(String)PostalCode=02192
(String)Phone=6175553985
(String)Status=A
(String)SocialSecurityNumber=017349033
(String)Salary=$45,700.00
(DateTime)StartDate=28/08/1984 0:00:00 AM
(DateTime)TerminationDate=(null)
(DateTime)BirthDate=05/06/1958 0:00:00 AM
(Boolean)BenefitHealthInsurance=True
(Boolean)BenefitLifeInsurance=True
(Boolean)BenefitDayCare=False
(String)Sex=F
```

There are a couple of things to note about the results.

- Columns that contain nulls are returned as type DBNull.
- The Salary amount has been converted to the client's currency format.
- Columns that contain date but no time values assume a time of 00:00:00 or midnight.

Tutorial: Using data types with JAX-WS

The following tutorial demonstrates how to access web services using the Java API for XML Web Services (JAX-WS).

If you have done the earlier JAX-WS tutorial, then some of these steps will already have been performed.

Before you begin, you will need the JAX-WS tools available from Sun. If you do not have this package installed on your system, you must download the JAX-WS tools and install them. To download the JAX-WS tools, visit http://java.sun.com/webservices/. Click the link for JAX-WS. This should take you to the Java API for XML Web Services page. Click the Download Now link. After you have downloaded the software package, install it on your system.

This example was developed with JAX-WS 2.1.3 for Windows.

SQL Anywhere SOAP web services that are accessed from JAX-WS should be declared to be of format CONCRETE.

To create SOAP and DISH services

1. At a command prompt, execute the following statement to start a personal web server. The `-xs http(port=80)` option tells the database server to accept HTTP requests on port 80. If you already have a web server running on port 80, use another port number such as 8080 for this tutorial.

   ```
   dbeng11 -xs http(port=80) samples-dir\demo.db
   ```

2. Start Interactive SQL and connect to the SQL Anywhere sample database as the DBA. Execute the following statements:

   a. Define a stored procedure that lists some columns of the Employees table.

   ```
   CREATE PROCEDURE ListEmployees()
   RESULT (
    EmployeeID            INTEGER,
    Surname               CHAR(20),
    GivenName             CHAR(20),
    StartDate             DATE,
    TerminationDate       DATE )
   BEGIN
     SELECT EmployeeID, Surname, GivenName,
       StartDate, TerminationDate
     FROM Employees;
   END;
   ```

   b. Define a SOAP service that calls this stored procedure.

   ```
   CREATE SERVICE "WS/EmployeeList2"
   TYPE 'SOAP'
   FORMAT 'CONCRETE' EXPLICIT ON
   DATATYPE OUT
   AUTHORIZATION OFF
   SECURE OFF
   USER DBA
   AS CALL ListEmployees();
   ```

   The EXPLICIT clause can only be used with a SOAP or DISH service of type CONCRETE. In this example, EXPLICIT ON indicates that the corresponding DISH service should generate XML Schema that describes the EmployeeList2Dataset object. This option only affects the WSDL

document that is generated. In an earlier JAX-WS tutorial, we looked at an example that used EXPLICIT OFF. See "Tutorial: Accessing web services from JAX-WS" on page 738.

DATATYPE OUT indicates that explicit data type information is generated in the XML result set response. If DATATYPE OFF had been specified, then all data would be typed as string. This option does not affect the WSDL document that is generated.

Because authorization has been turned off, anyone can use this service without supplying a user name and password. The commands run under user DBA. This arrangement is simple, but not secure.

c. Create a DISH service to act as a proxy for the SOAP service and to generate the WSDL document.

```
CREATE SERVICE "WSDish"
TYPE 'DISH'
FORMAT 'CONCRETE'
GROUP "WS"
AUTHORIZATION OFF
SECURE OFF
USER DBA;
```

The SOAP and DISH service must both be format CONCRETE. Since the EmployeeList2 service is in the WS group, a GROUP clause is included.

3. Take a look at the WSDL that the DISH service automatically creates. To do so, open a web browser and browse to the following URL: http://localhost:80/demo/WSDish. The DISH service automatically generates a WSDL document that appears in the browser window.

In particular, observe the EmployeeList2Dataset object that is exposed because the format of this service is CONCRETE and EXPLICIT is ON. In a later step, the **wsimport** application uses this information to generate a SOAP 1.1 client interface for this service.

```
<s:complexType name="EmployeeList2Dataset">
<s:sequence>
<s:element name="rowset">
  <s:complexType>
  <s:sequence>
  <s:element name="row" minOccurs="0" maxOccurs="unbounded">
    <s:complexType>
    <s:sequence>
    <s:element minOccurs="0" maxOccurs="1" name="EmployeeID"
        nillable="true" type="s:int" />
    <s:element minOccurs="0" maxOccurs="1" name="Surname"
        nillable="true" type="s:string" />
    <s:element minOccurs="0" maxOccurs="1" name="GivenName"
        nillable="true" type="s:string" />
    <s:element minOccurs="0" maxOccurs="1" name="StartDate"
        nillable="true" type="s:date" />
    <s:element minOccurs="0" maxOccurs="1" name="TerminationDate"
        nillable="true" type="s:date" />
    </s:sequence>
    </s:complexType>
  </s:element>
  </s:sequence>
  </s:complexType>
</s:element>
</s:sequence>
</s:complexType>
```


To generate a JAX-WS interface for these web services

1. In this example, the Java API for XML Web Services (JAX-WS) and the Sun Java 1.6.0 JDK are installed on drive *C:*. Set your PATH environment variable so that it includes the JAX-WS binaries, as well as the JDK binaries. The binaries are located in the following directories:

   ```
   c:\Sun\jaxws-ri\bin
   c:\jdk1.6.0\bin
   ```

2. Set your CLASSPATH environment variable.

   ```
   SET classpath=.;C:\Sun\jaxws-ri\lib\jaxb-api.jar
   ;C:\Sun\jaxws-ri\lib\jaxws-rt.jar
   ```

3. The next step is to generate the interface needed to call the web service.

   At a command prompt, execute the following command.

   ```
   wsimport -keep -Xendorsed "http://localhost:80/demo/WSDish"
   ```

   The wsimport tool retrieves the WSDL document from the given URL, generates the Java files that define the interface for it, and then compiles the Java files.

   The **keep** option tells wsimport not to delete the *.java* files. Without this option, these files are deleted after the corresponding *.class* files have been generated. Saving these files makes it easer to examine the makeup of the interface.

   The **Xendorsed** option allows you to use the JAX-WS 2.1 API with JDK6.

   Once this command completes, you should have a new subdirectory structure named *localhost\demo\ws* in your current directory that contains the following Java files, along with the compiled *.class* versions of each Java file.

   ```
   EmployeeList2.java
   EmployeeList2Dataset.java
   EmployeeList2Response.java
   FaultMessage.java
   ObjectFactory.java
   package-info.java
   WSDish.java
   WSDishSoapPort.java
   ```

To use the generated JAX-WS interface

1. Save the following Java source code into *SASoapDemo2.java*. Make sure that this file is located in the same directory containing the *localhost* subdirectory that was generated by the wsimport tool.

   ```
   // SASoapDemo2.java illustrates a web service client that
   // calls the WSDish service and prints out the data.

   import java.util.*;
   import javax.xml.ws.*;
   import org.w3c.dom.Element;
   import org.w3c.dom.Node;
   import javax.xml.datatype.*;
   import localhost.demo.ws.*;

   public class SASoapDemo2
   {
   ```

```
public static void main( String[] args )
{
  try {
    WSDish service = new WSDish();

    Holder<EmployeeList2Dataset> response =
        new Holder<EmployeeList2Dataset>();
    Holder<Integer> sqlcode = new Holder<Integer>();

    WSDishSoapPort port = service.getWSDishSoap();

    // This is the SOAP service call to EmployeeList2
    port.employeeList2( response, sqlcode );

    EmployeeList2Dataset result = response.value;
    EmployeeList2Dataset.Rowset rowset = result.getRowset();

    List<EmployeeList2Dataset.Rowset.Row> rows = rowset.getRow();

    String fieldType;
    String fieldName;
    String fieldValue;
    Integer fieldInt;
    XMLGregorianCalendar fieldDate;

    for ( int i = 0; i < rows.size(); i++ ) {
      EmployeeList2Dataset.Rowset.Row row = rows.get( i );

      fieldType = row.getEmployeeID().getDeclaredType().getSimpleName();
      fieldName = row.getEmployeeID().getName().getLocalPart();
      fieldInt = row.getEmployeeID().getValue();
      System.out.println( "(" + fieldType + ")" + fieldName +
                          "=" + fieldInt );

      fieldType = row.getSurname().getDeclaredType().getSimpleName();
      fieldName = row.getSurname().getName().getLocalPart();
      fieldValue = row.getSurname().getValue();
      System.out.println( "(" + fieldType + ")" + fieldName +
                          "=" + fieldValue );

      fieldType = row.getGivenName().getDeclaredType().getSimpleName();
      fieldName = row.getGivenName().getName().getLocalPart();
      fieldValue = row.getGivenName().getValue();
      System.out.println( "(" + fieldType + ")" + fieldName +
                           "=" + fieldValue );

      fieldType = row.getStartDate().getDeclaredType().getSimpleName();
      fieldName = row.getStartDate().getName().getLocalPart();
      fieldDate = row.getStartDate().getValue();
      System.out.println( "(" + fieldType + ")" + fieldName +
                          "=" + fieldDate );

      if ( row.getTerminationDate() == null ) {
        fieldType = "unknown";
        fieldName = "TerminationDate";
        fieldDate = null;
      } else {
        fieldType =
          row.getTerminationDate().getDeclaredType().getSimpleName();
        fieldName = row.getTerminationDate().getName().getLocalPart();
        fieldDate = row.getTerminationDate().getValue();
      }
      System.out.println( "(" + fieldType + ")" + fieldName +
                          "=" + fieldDate );
```



```
            System.out.println();
          }
        }
        catch (Exception x) {
          x.printStackTrace();
        }
      }
    }
```

If you chose to start the web server using a different port number such as 8080, then you will have to alter the `import localhost` source line to something like the following:

```
import localhost._8080.demo.ws.*;
```

2. Compile *SASoapDemo2.java*.

```
javac SASoapDemo2.java
```

3. Run the compiled class file.

```
java SASoapDemo2
```

When the application sends its request to the web server, it receives an XML result set response that consists of an EmployeeList2Result with a rowset containing several row entries. Also included in the response is the sqlcode result from executing the query. An example of the response is shown next.

```
<tns:EmployeeList2Response>
 <tns:EmployeeList2Result xsi:type='tns:EmployeeList2Dataset'>
  <tns:rowset>
    <tns:row>...</tns:row>
.
.
.
    <tns:row>...</tns:row>
   </tns:rowset>
 </tns:EmployeeList2Result>
 <tns:sqlcode>0</tns:sqlcode>
</tns:EmployeeList2Response>
```

Each row in the rowset is sent in a format similar to the following.

```
<tns:row>
   <tns:EmployeeID xsi:type="xsd:int">1751</tns:EmployeeID>
   <tns:Surname xsi:type="xsd:string">Ahmed</tns:Surname>
   <tns:GivenName xsi:type="xsd:string">Alex</tns:GivenName>
   <tns:StartDate xsi:type="xsd:date">1994-07-12-04:00</tns:StartDate>
   <tns:TerminationDate xsi:type="xsd:date">2008-04-18-04:00
       </tns:TerminationDate>
</tns:row>
```

Note that the column name and data type are included.

Using a proxy

You can observe the response shown above through the use of proxy software that logs the XML message traffic. The proxy inserts itself between your client application and the web server.

The EmployeeList2 result set is displayed as (type)column name=value pairs by the SASoapDemo2 application. Several lines of output similar to the following should be generated.

```
(Integer)EmployeeID=102
(String)Surname=Whitney
(String)GivenName=Fran
(XMLGregorianCalendar)StartDate=1984-08-28-04:00
(unknown)TerminationDate=null

(Integer)EmployeeID=105
(String)Surname=Cobb
(String)GivenName=Matthew
(XMLGregorianCalendar)StartDate=1985-01-01-05:00
(unknown)TerminationDate=null
.
.
.
(Integer)EmployeeID=1740
(String)Surname=Nielsen
(String)GivenName=Robert
(XMLGregorianCalendar)StartDate=1994-06-24-04:00
(unknown)TerminationDate=null

(Integer)EmployeeID=1751
(String)Surname=Ahmed
(String)GivenName=Alex
(XMLGregorianCalendar)StartDate=1994-07-12-04:00
(XMLGregorianCalendar)TerminationDate=2008-04-18-04:00
```

Note that the TerminationDate column is only sent when its value is not NULL. The Java application is designed to detect when the TerminationDate column is not present. For this example, the last row in the Employees table was altered such that a non-NULL termination date was set.

Also note that date values include an offset from UTC time. In the above sample data, the server is located in the North American Eastern time zone. This is 5 hours earlier than UTC time (-05:00) for dates when daylight savings is not in effect and 4 hours earlier than UTC time (-04:00) for dates when daylight savings is in effect.

For further understanding of the Java methods used in the SASoapDemo2 application, it is worthwhile investigating the javax.xml.bind.JAXBElement class documentation on the java.sun.com web site (http://java.sun.com/javaee/5/docs/api/).

Using the iAnywhere WSDL compiler

Given a WSDL source that describes a web service, the iAnywhere WSDL compiler generates a set of Java proxy classes, C# proxy classes, or SQL SOAP client procedures for SQL Anywhere that you include in your application.

The Java or C# classes generated by the WSDL compiler are intended for use with QAnywhere. These classes expose web service operations as method calls. The classes that are generated are:

- The main service binding class (this class inherits from ianywhere.qanywhere.ws.WSBase in the mobile web services runtime).
- A proxy class for each complex type specified in the WSDL file.

For information about the generated proxy classes, see:

- .NET: "iAnywhere.QAnywhere.WS namespace (.NET 2.0)" [*QAnywhere*]
- Java: "ianywhere.qanywhere.ws package" [*QAnywhere*]

The WSDL compiler supports WSDL 1.1 and SOAP 1.1 over HTTP and HTTPS.

Syntax

wsdlc [*options*] *wsdl-uri*

wsdl-uri:

This is the specification for the WSDL (Web Services Description Language) source (a URL or file).

Options:

- **-h** Display help text.
- **-v** Display verbose information.
- **-o *output-directory*** Specify an output directory for generated files.
- **-l *language*** Specify a language for the generated files. This is one of **java**, **cs** (for C#), or **sql**. These options must be specified in lowercase letters.
- **-d** Display debug information that may be helpful when contacting iAnywhere customer support.

Java-specific options:

- **-p *package*** Specify a package name. This permits you to override the default package name.

C#-specific options:

- **-n *namespace*** Specify a namespace. This permits you to wrap the generated classes in a namespace of your choosing.

SQL-specific options:

- **-f *filename*** (Required) Specify the name of the output SQL file to which the SQL statements are written. This operation overwrites any existing file of the same name.
- **-p=*prefix*** Specify a prefix for the generated function or procedure names. The default prefix is the service name followed by a period (for example, "WSDish.").

- **-x** Generate procedure definitions rather than function definitions.

This is the name of the WSDL file that describes a web service.

WSDLC does not expand complex parameters representing structures or arrays. Such parameters are commented out in order to allow the database server to automatically create the given procedure or function without modification. However, in order for the SOAP operation to work, such parameters must be analyzed and manually composed. The process may require using the SQL Anywhere XMLELEMENT function with the XMLATTRIBUTES parameter to generate complex XML representations of structures.

Creating web service client functions and procedures

As well as providing web services, a SQL Anywhere database can consume web services. These can be standard web services available over the Internet, or these can be provided by SQL Anywhere databases, as long as the web service is not in the same database as the client procedure or function.

SQL Anywhere can act as both HTTP and SOAP web service clients. This functionality is provided through stored functions and stored procedures.

Client functions and procedures are created and manipulated using the following SQL statements:

- "CREATE FUNCTION statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER FUNCTION statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*]
- "DROP statement" [*SQL Anywhere Server - SQL Reference*]

For example, the syntax of the CREATE FUNCTION and CREATE PROCEDURE statements, as used to create web service client functions, are as follows:

CREATE FUNCTION [*owner*.]*procedure-name* **(** [*parameter*, ...] **)**
RETURNS *data-type*
URL *url-string*
[*proc-attributes*]

CREATE PROCEDURE [*owner*.]*procedure-name* **(** [*parameter*, ...] **)**
URL *url-string*
[*proc-attributes*]

Key to this syntax is the URL clause, which is used to provide the URL of the web service that you want the procedure to access. The basic syntax of the URL clause is as follows:

url-string :
'{ HTTP | **HTTPS** | **HTTPS_FIPS** }**://**[*user***:***password***@**]*hostname*[**:***port*][**/***path*]**'**

The optional user and password information permits you to access web services that require authentication. The hostname can be the name or the IP address of the computer providing the web service.

The port number is required only if the server is listening on a port number other than the default. The default port numbers are 80 for HTTP services and 443 for HTTPS services.

The path identifies the resource or web service on the server.

A request can be sent to any web service, whether it is provided by another SQL Anywhere database or available over the Internet. If the web service is provided by the same database server, it cannot reside in the same database as the client function. Attempting to access a web service in the same database results in the error 403 Forbidden.

Because it is used for parameter substitution, exclamation marks that appear in strings anywhere in the procedure definition must be escaped. For more information, see "Escaping the ! character" on page 775.

Common clauses

The additional clauses that enable you to supply more detailed information about the procedure call are as follows:

proc-attributes :
[**TYPE** { **'HTTP**[**:**{ **GET** | **POST**[**:***MIME-type*] | **PUT**[**:***MIME-type*] | **DELETE** | **HEAD** }]' |
'SOAP[**:**{ **RPC** | **DOC** }]' }]
[**NAMESPACE** *namespace-string*]
[**CERTIFICATE** *certificate-string*]
[**CLIENTPORT** *clientport-string*]
[**PROXY** *proxy-string*]

The TYPE clause is important because it tells SQL Anywhere how to format the request to the web service provider. Standard SOAP types RPC and DOC are available. The standard HTTP methods of GET and POST are available, specified as HTTP:GET and HTTP:POST, respectively. Specifying HTTP implies HTTP:POST.

If type SOAP is chosen, SQL Anywhere automatically formats the request as an XML document in the standard format necessary for SOAP requests. Since a SOAP request is always an XML document, an HTTP POST request is always implicitly used to send the SOAP request document to the server when type SOAP is chosen. Specifying SOAP implies SOAP:RPC.

Names for web service client functions and procedures

The procedure name is used as the SOAP operation name when building the outgoing SOAP request. In addition, the names of any parameters also appear in tagnames in the SOAP request envelope. Since the SOAP server expects to see these names, specifying them correctly is an important part of defining a SOAP stored procedure. This fact places restrictions on the name of SOAP procedures and functions, beyond the general rules that apply to procedure and function names in SQL Anywhere.

The following statement creates a SOAP stored procedure named MyOperation:

```
CREATE PROCEDURE MyOperation ( a INTEGER, b CHAR(128) )
URL 'HTTP://localhost'
TYPE 'SOAP:DOC';
```

When this procedure is called, such as by the following statement, a SOAP request is generated:

```
CALL MyOperation( 123, 'abc' );
```

The procedure name appears in the `<m:MyOperation>` tag within the request body. The two parameters to the procedure, a and b, become `<m:a>` and `<m:b>`, respectively.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:m="http://localhost">
  <SOAP-ENV:Body>
    <m:MyOperation>
      <m:a>123</m:a>
      <m:b>abc</m:b>
    </m:MyOperation>
  </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```


Namespace URIs

All SOAP requests require a method namespace URI. The SOAP processor on the server side uses this URI to understand the names of the various entities in the message body of the request.

When creating a SOAP function or procedure, of either SOAP:DOC or SOAP:RPC, you may be required to specify a namespace URI before the call succeeds. You can obtain the required namespace value from the WSDL description document, or from other available documentation for the service. The NAMESPACE clause applies only to SOAP functions and procedures. The default namespace value is the procedure's URI, up to, but not including, the optional path component and excluding any user and password values.

HTTPS requests

To issue a secure HTTPS request, the client must have access to the server's certificate, or the certificate used to sign the server's certificate. This certificate tells SQL Anywhere how to encrypt the request. Certificate values are also required when a request directed to an insecure server might be redirected to a secure one.

There are two ways to provide the certificate information. You can either place the certificate in a file and provide the file name, or you can provide the entire certificate as a string value. You cannot do both.

The certificate attributes are supplied as a string value constructed as key=value pairs separated by semicolons:

certificate-string :
{ **file=***filename* | **certificate=***string* } **; company=***company* **; unit=***company-unit* **; name=** *common-name*

The following keys are available:

Key	Abbreviation	Description
file		The file name of the certificate.
certificate	cert	The certificate itself, Base64 encoded.
company	co	The company specified in the certificate.
unit		The company unit specified in the certificate.
name		The common name specified in the certificate.

For example, the following statement creates a procedure that makes a secure request to a web service located on the same computer as the client:

```
CREATE PROCEDURE test()
URL 'HTTPS://localhost/myservice'
CERTIFICATE 'file=C:\srv_cert.id;co=iAnywhere;
             unit=SA;name=JohnSmith';
```

Since no TYPE clause was provided, the request is assumed to be of type SOAP:RPC. The server's public certificate is located in the file *C:\srv_cert.id*.

Client ports

When accessing web services through a firewall, it is sometimes necessary to tell SQL Anywhere which ports to use when opening a connection to the server. Ordinarily, port numbers are obtained dynamically, and you should rely on the default behavior unless your firewall restricts access to a particular range of ports.

The ClientPort option designates the port number on which the client application communicates using TCP/IP. You can specify a single port number, or a combination of individual port numbers, and ranges of port numbers, as demonstrated in the following example:

```
CREATE PROCEDURE test ()
URL 'HTTPS://localhost/myservice'
CLIENTPORT '5040,5050-5060,5070';
```

It is best to specify a list or a range of port numbers. If you specify a single port number, then only one connection is maintained at a time. In fact, even after closing the one connection, there is a timeout period of several minutes during which no new connection can be made to the same remote server and port. When you specify a list and/or range of port numbers, the application keeps trying port numbers until it finds one to which it can successfully bind.

This feature is similar to the ClientPort network protocol option. For more information, see "ClientPort protocol option [CPORT]" [*SQL Anywhere Server - Database Administration*].

Using proxies

Some web service requests may need to be made through a proxy server. When this is the case, the URL of the proxy server must be supplied using the PROXY clause.

The format of the value is the same as for the URL clause, although any user, password, or path values are ignored:

proxy-string :
'{ **HTTP** | **HTTPS** }**://**[*user***:***password***@**]*hostname*[**:***port*][**/***path*]'

When a proxy server is specified, SQL Anywhere formats the request and sends it to the proxy server using the supplied proxy URL. The proxy server forwards the request to the final destination, obtains the response, and forwards the response back to SQL Anywhere.

Logging web service client procedures

Information from web service clients, including HTTP requests and transport data, can be logged to the **web service client log file**. You enable logging to this file by starting the database server with the -zoc server option or by setting the WebClientLogging server property using the sa_server_option system procedure. See "-zoc server option" [*SQL Anywhere Server - Database Administration*], and "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*].

Modifying HTTP headers

When creating a web service procedure using the CREATE PROCEDURE statement, if you specify an HTTP HEADER name with no colon (:) and no value, the HTTP client application suppresses the header. If you include a colon but do not provide a value, the HTTP client application includes the header name, but no value. For example:


```
CREATE PROCEDURE suds(...)
TYPE 'SOAP:RPC'
URL '...'
HEADER 'SOAPAction\nDate\nFrom:';
```

In this example, the Action and Date HTTP headers, which are automatically generated by SQL Anywhere, are suppressed, and the From header is included but with no value.

Modifying automatically generated headers, can have unexpected results. The following HTTP headers are typically automatically generated and should not be modified without very careful consideration.

HTTP header	Description
Accept-Charset	Always automatically generated. If changed or deleted, may result in unexpected data conversion errors.
ASA-Id	Always automatically generated. Ensures client does not connect to itself (that is to the same server) which may result in deadlock.
Authorization	Automatically generated when URL contains credentials. If changed or deleted, may result in failure of the request. Only BASIC authorization is supported. User and password information should only be included when connecting via HTTPS.
Connection	Connection: close, is always automatically generated. Client does not support persistent connections. Should not be changed or connection may hang.
Host	Always automatically generated. HTTP/1.1 servers are required to respond with 400 Bad Request if an HTTP/1.1 client does not provide a Host header.
Transfer-Encoding	Automatically generated when posting a request in chunk mode. Removing this header or deleting the chunked value will result in failure when the client is using CHUNK mode.
Content-Length	Automatically generated when posting a request and not in chunk mode. This header is required to tell the server the content length of the body. If the content length is wrong the connection may hang or data loss could occur.

Working with return values and result sets

Web service client calls can be made with either stored functions, or stored procedures. If made from a function, the return type of the function must be of a character data type, such as CHAR, VARCHAR, or LONG VARCHAR. The value returned is the body of the HTTP response. No header information is included. Additional information about the request, including the HTTP status information, is returned by procedures. Thus, procedures are preferred when access to this additional information is wanted.

SOAP procedures

The response from a SOAP function is an XML document that contains the SOAP response.

Since SOAP responses are structured XML documents, SQL Anywhere by default attempts to exploit this information and construct a more useful result set. Each of the top-level tags within the returned response document is extracted and used as a column name. The contents of the subtree below each of these tags is used as the row value for that column.

For example, given the SOAP response shown below, SQL Anywhere would construct the shown data set:

```
<SOAP-ENV:Envelope
  xmlns:SOAPSDK1="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAPSDK2="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAPSDK3="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
     <ElizaResponse xmlns:SOAPSDK4="SoapInterop">
        <Eliza>Hi, I'm Eliza. Nice to meet you.</Eliza>
     <ElizaResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Eliza
Hi, I'm Eliza. Nice to meet you.

In this example, the response document is delimited by the ElizaResponse tags that appear within the SOAP-ENV:Body tags.

Result sets have as many columns as there are top-level tags. This result set has only one column because there is only one top-level tag in the SOAP response. This single top-level tag, Eliza, becomes the name of the column.

XML processing facilities

Information within XML result sets, including SOAP responses, can also be accessed using the built-in Open XML processing capabilities.

The following example uses the OPENXML procedure to extract portions of a SOAP response. This example uses a web service to expose the contents of the SYSWEBSERVICE table as a SOAP service:

```
CREATE SERVICE get_webservices
TYPE 'SOAP'
AUTHORIZATION OFF
USER DBA
AS SELECT * FROM SYSWEBSERVICE;
```


The following stored function, which must be created in a second SQL Anywhere database, issues a call to this web service. The return value of this function is the entire SOAP response document. The response is in the .NET DataSet format, as DNET is the default SOAP service format.

```
CREATE FUNCTION get_webservices()
RETURNS LONG VARCHAR
URL 'HTTP://localhost/get_webservices'
TYPE 'SOAP:DOC';
```

The following statement demonstrates how two columns of the result set can be extracted with the aid of the OPENXML procedure. These are the *service_name* and *secure_required* columns, which indicate which SOAP services are secure and hence require HTTPS.

```
SELECT *
FROM openxml( get_webservices(), '//row' )
WITH ("Name"    char(128) 'service_name',
      "Secure?" char(1)   'secure_required' );
```

This statement works by selecting the decedents of the *row* node. The WITH clause constructs the result set based on the two elements of interest. Assuming only the *get_webservices* service exists, this function returns the following result set:

Name	Secure?
get_webservices	N

For more information about the XML processing facilities available in SQL Anywhere, see "Using XML in the database" [*SQL Anywhere Server - SQL Usage*].

Other types of procedures

Procedures of other types return all the information about a response in a two-column result set. This result set includes the response status, header information and body. The first column, is named Attribute and the second Value. Both are of data type LONG VARCHAR.

The result set has one row for each of the response header fields, as well as a row for the HTTP status line (Status attribute) and a row for the response body (Body attribute).

The following example represents a typical response:

Attribute	Value
Status	HTTP /1.0 200 OK
Body	<!DOCTYPE HTML ... ><HTML> ... </HTML>
Content-Type	text/html
Server	GWS/2.1
Content-Length	2234
Date	Mon, 18 Oct 2004, 16:00:00 GMT

Selecting from result sets

The SELECT statement is used to retrieve values from results sets. Once retrieved, these values can be stored in tables or used to set variables.

```
CREATE PROCEDURE test( INOUT parm CHAR(128) )
URL 'HTTP://localhost/test'
TYPE 'HTTP';
```

Because it is of type HTTP, this procedure returns the two-column result set described in the previous section. In the first column is an attribute name; in the second column the attribute value. The keywords are as in the HTTP response header fields. A Body attribute contains the body of the message, which is typically an HTML document.

One approach is to insert the result sets into a table, such as the following one:

```
CREATE TABLE StoredResults(
     Attribute LONG VARCHAR,
     Value     LONG VARCHAR
);
```

Result sets can be inserted into this table as follows:

```
INSERT INTO StoredResults SELECT *
FROM test('Storing into a table')
WITH (Attribute LONG VARCHAR, Value LONG VARCHAR);
```

You can add clauses according to the usual syntax of the SELECT statement. For example, if you want only a specific row of the result set you can add a WHERE clause to limit the results of the select to only one row:

```
SELECT Value
FROM test('Calling test for the Status Code')
WITH (Attribute LONG VARCHAR, Value LONG VARCHAR)
WHERE Attribute = 'Status';
```

This statement selects only the status information from the result set. It can thus be used to verify that the call was successful.

Using parameters

Stored functions and stored procedures that act as web service clients can be declared with parameters, just like other types of function or procedures. These parameter values, unless used during parameter substitution, are passed as part of the HTTP or SOAP request.

In addition, parameters can be used to replace placeholders within the body of the stored function or stored procedure at the time the function or procedure is called. If no placeholders for a particular variable exist, the parameter and its value are passed as part of the request. Parameters and values used for substitution in this manner are not passed as part of the web service request.

Passed parameters

All parameters to a function or procedure, unless used during parameter substitution, are passed as part of the web service request. The format in which they are passed depends on the type of the web service request.

HTTP requests

Parameters to requests of type HTTP:GET are encoded in the URL. For example, the following procedure declares two parameters:

```
CREATE PROCEDURE test ( a INTEGER, b CHAR(128) )
URL 'HTTP://localhost/myservice'
TYPE 'HTTP:GET';
```

If this procedure is invoked with the two values 123 and 'xyz', then the URL used for the request is equivalent to that shown below:

```
HTTP://localhost/myservice?a=123&b=xyz
```

If the type is HTTP:POST, the parameters and their values instead become part of the body of the request. In the case of the two parameter and values, the following text appears, after the headers, in the body of the HTTP request:

```
a=123&b=xyz
```

SOAP requests

Parameters passed to SOAP requests are bundled as part of the request body, as required by the SOAP specification:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:m="http://localhost">
  <SOAP-ENV:Body>
    <m:test>
      <m:a>123</m:a>
      <m:b>abc</m:b>
    </m:test>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Parameter substitution

Declared parameters to the stored procedure or stored function are automatically substituted for placeholders within the stored function or stored procedure definition each time the procedure or function is executed. Any substrings that contain an exclamation mark (!) followed by the name of one of the declared parameters are replaced by that parameter's value.

For example, the following procedure definition permits the entire URL to be passed as a parameter. Different values can be used each time this procedure is called.

```
CREATE PROCEDURE test ( url CHAR(128) )
URL '!url'
TYPE 'HTTP:POST';
```

For example, you could then use the procedure as follows:

```
CALL test ( 'HTTP://localhost/myservice' );
```

Hiding user and password values

One useful application of parameter substitution is to avoid making sensitive values, such as user names and passwords, part of a web service client function or procedure definition. When such values are specified as literals in the procedure or function definition, they are stored in the system tables, making them readily accessible to all users of the database. Passing these values as parameters circumvents this problem.

For example, the following procedure definition includes the user name and password as plain text as part of the procedure definition:

```
CREATE PROCEDURE test
URL 'HTTP://dba:sql@localhost/myservice';
```

To avoid this problem, you can declare user and password as parameters. Doing so permits the user and password values to be supplied only when the procedure is invoked. For example:

```
CREATE PROCEDURE test ( uid CHAR(128), pwd CHAR(128) )
URL 'HTTP://!uid:!pwd@localhost/myservice';
```

This procedure is called as follows:

```
CALL test ( 'dba', 'sql' );
```

As another example, you can use parameter substitution to pass encryption certificates from a file and pass them to a stored procedure or stored function:

```
CREATE PROCEDURE secure( cert LONG VARCHAR )
URL 'https://localhost/secure'
TYPE 'HTTP:GET'
CERTIFICATE 'cert=!cert;company=test;unit=test;name=RSA Server';
```

When you call this procedure, you supply the certificate as a string. In the following example call, the certificate is read from a file. This is done for illustration only, as the certificate can be read directly from a file using the **file=** keyword for the CERTIFICATE clause.

```
CALL secure( xp_read_file('install-dir\bin32\rsaserver.id') );
```


Escaping the ! character

Because the exclamation mark (!) is used to identify placeholders for parameter substitution in the context of web service client stored functions and stored procedures, it must be escaped whenever you want to include this character as part of any of the procedure attribute strings string. To do so, prefix the exclamation mark with a second exclamation mark. Hence, all occurrences of !! in strings within a web service client or web service function definition are replaced by !.

Parameter names used as placeholders must contain only alphanumeric characters. In addition, placeholders must be followed by a non-alphanumeric character to avoid ambiguity. Placeholders with no matching parameter name are automatically deleted. For example, the parameter size would not be substituted for the placeholder in the following procedure:

```
CREATE PROCEDURE orderitem ( size CHAR(18) )
URL 'HTTP://salesserver/order?size=!sizeXL'
TYPE 'SOAP:RPC';
```

Instead, !sizeXL is always deleted because it is a valid placeholder for which there is no matching parameter.

Parameter data type conversion

Parameter values that are not of character or binary data types are converted to a string representation before being added to the request. This process is equivalent to casting the value to a character type. The conversion is done in accordance with the data type formatting option settings at the time the function or procedure is invoked. In particular, the conversion can be affected by such options as precision, scale, and timestamp_format.

Working with structured data types

XML return values

The SQL Anywhere server as a web service client may interface to a web service using a function or a procedure.

For simple return data types, a string representation within a result set may suffice. In such a case, the use of a stored procedure may be warranted.

The use of web service functions are a better choice when returning complex data such as arrays or structures. For function declarations, the RETURN clause can specify an XML data type. The returned XML can be parsed using OPENXML to extract the elements of interest.

Note that a return of XML data such as dateTime will be rendered within the result set verbatim. For example, if a TIMESTAMP column was included within a result set, it would be formatted as an XML dateTime string (2006-12-25T12:00:00.000-05:00) not as a string (2006-12-25 12:00:00.000).

XML parameter values

The SQL Anywhere XML data type is supported for use as a parameter within web service functions and procedures. For simple types, the parameter element is automatically constructed when generating the SOAP request body. However, for parameters of type XML, this cannot be done since the XML representation of the element may require attributes that provide additional data. Therefore, when generating the XML for a parameter whose data type is XML, the root element name must correspond to the parameter name.

```
<inputHexBinary xsi:type="xsd:hexBinary">414243</inputHexBinary>
```

The XML type demonstrates how to send a parameter as a hexBinary XML type. The SOAP endpoint expects that the parameter name (or in XML terms, the root element name) is "inputHexBinary".

Cookbook constants

Knowledge of how SQL Anywhere references namespaces is required to construct complex structures and arrays. The prefixes listed here correspond to the namespace declarations generated for a SQL Anywhere SOAP request envelope.

SQL Anywhere XML Prefix	Namespace
xsd	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
SOAP-ENC	http://schemas.xmlsoap.org/soap/encoding/
m	namespace as defined in the NAMESPACE clause

Complex data type examples

The following three examples demonstrate how to create web service client functions taking parameters that represent an array, a structure, and an array of structures. The examples are designed to issue requests to the Microsoft SOAP ToolKit 3.0 Round 2 Interoperability test server (http://mssoapinterop.org/stkV3). The web service functions will communicate to SOAP operations (or RPC function names) named echoFloatArray,

echoStruct, and echoStructArray respectively. The common namespace used for Interoperability testing is "http://soapinterop.org/", allowing a given function to test against alternative Interoperability servers simply by changing the URL clause to the selected SOAP endpoint.

All three examples use a table to generate the XML data. The following shows how to set up that table.

```
CREATE LOCAL TEMPORARY TABLE SoapData
(
    seqno INT DEFAULT AUTOINCREMENT,
    i INT,
    f FLOAT,
    s LONG VARCHAR
) ON COMMIT PRESERVE ROWS;

INSERT INTO SoapData (i,f,s)
VALUES (99,99.999,'Ninety-Nine');

INSERT INTO SoapData (i,f,s)
VALUES (199,199.999,'Hundred and Ninety-Nine');
```

The following three functions send SOAP requests to the Interoperability server. Note that this sample issues requests to Microsoft's Interop server:

```
CALL sa_make_object('function', 'echoFloatArray');
ALTER FUNCTION echoFloatArray( inputFloatArray XML )
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';

CALL sa_make_object('function', 'echoStruct');
ALTER FUNCTION echoStruct( inputStruct XML )
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';

CALL sa_make_object('function', 'echoStructArray');
ALTER FUNCTION echoStructArray( inputStructArray XML )
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';
```

Finally, the three example statements along with the XML representation of their parameters are presented:

1. The parameters in the following example represent an array.

   ```
   SELECT echoFloatArray(
       XMLELEMENT( 'inputFloatArray',
          XMLATTRIBUTES( 'xsd:float[]' as "SOAP-ENC:arrayType" ),
           (
           SELECT XMLAGG( XMLELEMENT( 'number', f ) ORDER BY seqno )
           FROM SoapData
           )
       )
   );
   ```

   The stored procedure echoFloatArray will send the following XML to the Interoperability server.

   ```
   <inputFloatArray SOAP-ENC:arrayType="xsd:float[2]">
   <number>99.9990005493164</number>
   ```

```
<number>199.998992919922</number>
</inputFloatArray>
```

The response from the Interoperability server is shown below.

```
'<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope
  xmlns:SOAPSDK1="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAPSDK2="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAPSDK3="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAPSDK4:echoFloatArrayResponse
      xmlns:SOAPSDK4="http://soapinterop.org/">
      <Result SOAPSDK3:arrayType="SOAPSDK1:float[2]"
        SOAPSDK3:offset="[0]"
        SOAPSDK2:type="SOAPSDK3:Array">
        <SOAPSDK3:float>99.9990005493164</SOAPSDK3:float>
        <SOAPSDK3:float>199.998992919922</SOAPSDK3:float>
      </Result>
    </SOAPSDK4:echoFloatArrayResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>'
```

If the response was stored in a variable, then it can be parsed using OPENXML.

```
SELECT * FROM openxml( resp,'//*:Result/*' )
WITH ( varFloat FLOAT 'text()' );
```

varFloat
99.9990005493
199.9989929199

2. The parameters in the following example represent a structure.

```
SELECT echoStruct(
    XMLELEMENT('inputStruct',
        (
        SELECT XMLFOREST( s as varString,
                          i as varInt,
                          f as varFloat )
        FROM SoapData
        WHERE seqno=1
        )
    )
);
```

The stored procedure echoStruct will send the following XML to the Interoperability server.

```
<inputStruct>
  <varString>Ninety-Nine</varString>
  <varInt>99</varInt>
  <varFloat>99.9990005493164</varFloat>
</inputStruct>
```

The response from the Interoperability server is shown below.


```
'<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope
  xmlns:SOAPSDK1="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAPSDK2="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAPSDK3="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAPSDK4:echoStructResponse
      xmlns:SOAPSDK4="http://soapinterop.org/">
      <Result href="#id1"/>
    </SOAPSDK4:echoStructResponse>
    <SOAPSDK5:SOAPStruct
      xmlns:SOAPSDK5="http://soapinterop.org/xsd"
      id="id1"
      SOAPSDK3:root="0"
      SOAPSDK2:type="SOAPSDK5:SOAPStruct">
      <varString>Ninety-Nine</varString>
      <varInt>99</varInt>
      <varFloat>99.9990005493164</varFloat>
    </SOAPSDK5:SOAPStruct>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>'
```

If the response was stored in a variable, then it can be parsed using OPENXML.

```
SELECT * FROM openxml( resp,'//*:Body/*:SOAPStruct' )
WITH (
varString LONG VARCHAR 'varString',
varInt INT 'varInt',
varFloat FLOAT 'varFloat' );
```

varString	varInt	varFloat
Ninety-Nine	99	99.9990005493

3. The parameters in the following example represent an array of structures.

```
SELECT echoStructArray(
    XMLELEMENT( 'inputStructArray',
        XMLATTRIBUTES( 'http://soapinterop.org/xsd' AS "xmlns:q2",
                       'q2:SOAPStruct[2]' AS "SOAP-ENC:arrayType" ),
            (
            SELECT XMLAGG(
                XMLElement('q2:SOAPStruct',
                XMLFOREST( s as varString,
                           i as varInt,
                           f as varFloat )
                )
            ORDER BY seqno
            )
        FROM SoapData
        )
    )
);
```

The stored procedure echoFloatArray will send the following XML to the Interoperability server.

```
<inputStructArray xmlns:q2="http://soapinterop.org/xsd"
  SOAP-ENC:arrayType="q2:SOAPStruct[2]">
  <q2:SOAPStruct>
    <varString>Ninety-Nine</varString>
```

```
    <varInt>99</varInt>
    <varFloat>99.9990005493164</varFloat>
  </q2:SOAPStruct>
  <q2:SOAPStruct>
    <varString>Hundred and Ninety-Nine</varString>
    <varInt>199</varInt>
    <varFloat>199.998992919922</varFloat>
  </q2:SOAPStruct>
</inputStructArray>
```

The response from the Interoperability server is shown below.

```
'<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope
  xmlns:SOAPSDK1="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAPSDK2="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAPSDK3="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAPSDK4:echoStructArrayResponse
      xmlns:SOAPSDK4="http://soapinterop.org/">
      <Result xmlns:SOAPSDK5="http://soapinterop.org/xsd"
        SOAPSDK3:arrayType="SOAPSDK5:SOAPStruct[2]"
        SOAPSDK3:offset="[0]" SOAPSDK2:type="SOAPSDK3:Array">
        <SOAPSDK5:SOAPStruct href="#id1"/>
        <SOAPSDK5:SOAPStruct href="#id2"/>
      </Result>
    </SOAPSDK4:echoStructArrayResponse>
    <SOAPSDK6:SOAPStruct
      xmlns:SOAPSDK6="http://soapinterop.org/xsd"
      id="id1"
      SOAPSDK3:root="0"
      SOAPSDK2:type="SOAPSDK6:SOAPStruct">
      <varString>Ninety-Nine</varString>
      <varInt>99</varInt>
      <varFloat>99.9990005493164</varFloat>
    </SOAPSDK6:SOAPStruct>
    <SOAPSDK7:SOAPStruct
      xmlns:SOAPSDK7="http://soapinterop.org/xsd"
      id="id2"
      SOAPSDK3:root="0"
      SOAPSDK2:type="SOAPSDK7:SOAPStruct">
      <varString>Hundred and Ninety-Nine</varString>
      <varInt>199</varInt>
      <varFloat>199.998992919922</varFloat>
    </SOAPSDK7:SOAPStruct>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>'
```

If the response was stored in a variable, then it can be parsed using OPENXML.

```
SELECT * FROM openxml( resp,'//*:Body/*:SOAPStruct' )
WITH (
varString LONG VARCHAR 'varString',
varInt INT 'varInt',
varFloat FLOAT 'varFloat' );
```

varString	varInt	varFloat
Ninety-Nine	99	99.9990005493

varString	varInt	varFloat
Hundred and Ninety-Nine	199	199.9989929199

Working with variables

Variables in HTTP requests come from one of two sources. First, the URL can include a query string, which includes various name=value pairs. HTTP GET requests are formatted in this manner. Here is an example of a URL that contains a query string.

```
http://localhost/gallery?picture=sunset.jpg
```

The second way is through the URL path. Setting URL PATH to either ON or ELEMENTS causes the portion of the path following the service name to be interpreted as variable values. This option allows URLs to appear to be requesting a file in a particular directory, as would be the case on a traditional file-based web site, rather than something stored inside a database. The following is an example.

```
http://localhost/gallery/sunset.jpg
```

This URL appears to request the file *sunset.jpg* from a directory named gallery. In fact, the gallery service receives this string as a parameter (using it, perhaps, to retrieve a picture from a database table).

The parameter passed in HTTP requests depends on the setting of URL PATH.

- **OFF** No path parameters are permitted after the service name.
- **ON** All path elements after the service name are assigned to the variable URL.
- **ELEMENTS** The remainder of the URL path is split at the slash characters into a list of up to 10 elements. These values are assigned the variables URL1, URL2, URL3, ..., URL10. If there are fewer than 10 values, the remaining variables are set to NULL. Specifying more than ten variables causes an error.

Apart from the location in which they are defined, there is no difference between variables. You access and use all HTTP variables the same way. For example, the values of variables such as url1 are accessed in the same way as parameters that appear as part of a query, such as **?picture=sunset.jpg**.

Accessing variables

There are two main ways to access variables. The first is to mention variables in the statement of the service declaration. For example, the following statement passes the value of multiple variables to the ShowTable stored procedure:

```
CREATE SERVICE ShowTable
TYPE 'RAW'
AUTHORIZATION ON
AS CALL ShowTable( :user_name, :table_name, :limit, :start );
```

The other way is to use the built-in functions NEXT_HTTP_VARIABLE and HTTP_VARIABLE within the stored procedure that handles the request. If you do not know which variables are defined, you can use the NEXT_HTTP_VARIABLE to find out. The HTTP_VARIABLE function returns the variable values.

The NEXT_HTTP_VARIABLE function allows you to iterate through the names of the defined variables. The first time you call it, you pass in the NULL value. This returns the name of one variable. Calling it subsequent times, each time passing in the name of the previous variable, returns the next variable name. When the name of the last variable is passed to this function, it returns NULL.

Iterating through the variable names in this manner guarantees that each variable name is returned exactly once. However, the order that the values are returned may not be the same as the order that they appear in the request. In addition, if you iterate through the names a second time, they can be returned in a different order.

To get the value of each variable, use the HTTP_VARIABLE function. The first parameter is the name of the variable. Additional parameters are optional. In the case that multiple values were supplied for a variable, the function returns the first value if supplied with only one parameter. Supplying an integer as the second parameter allows you to retrieve additional values.

The third parameter allows you to retrieve variable header-field values from multi-part requests. Supply the name of a header field to retrieve its value. For example, the following SQL statements retrieve three variable values, then retrieve the header-field values of the image variable.

```
SET v_id = HTTP_VARIABLE( 'ID' );
SET v_title = HTTP_VARIABLE( 'Title' );
SET v_descr = HTTP_VARIABLE( 'descr' );

SET v_name = HTTP_VARIABLE( 'image', NULL, 'Content-Disposition' );
SET v_type = HTTP_VARIABLE( 'image', NULL, 'Content-Type' );
SET v_image = HTTP_VARIABLE( 'image', NULL, '@BINARY' );
```

Here is an example that uses the HTTP_VARIABLE function to retrieve the values associated with the variables. It is a modified version of the ShowSalesOrderDetail service described in an earlier section.

```
CREATE PROCEDURE ShowDetail()
BEGIN
    DECLARE v_customer_id LONG VARCHAR;
    DECLARE v_product_id LONG VARCHAR;
    SET v_customer_id = HTTP_VARIABLE( 'customer_id' );
    SET v_product_id = HTTP_VARIABLE( 'product_id' );
    CALL ShowSalesOrderDetail( v_customer_id, v_product_id );
END;
```

The service that invokes the stored procedure follows:

```
CREATE SERVICE ShowDetail
TYPE 'HTML'
URL PATH OFF
AUTHORIZATION OFF
USER DBA
AS CALL ShowDetail();
```

To test the service, open a web browser and supply the following URL:

http://localhost:80/demo/ShowDetail?product_id=300&customer_id=101

For more information about parameter passing, see "Understanding how URLs are interpreted" on page 728 and "Web services functions" [*SQL Anywhere Server - SQL Reference*].

Working with HTTP headers

Headers in HTTP requests can be obtained using a combination of the NEXT_HTTP_HEADER and HTTP_HEADER functions. The NEXT_HTTP_HEADER function iterates through the HTTP headers included within a request and returns the next HTTP header name. Calling it with NULL causes it to return the name of the first header. Subsequent headers are retrieved by passing the function the name of the previous header. This function returns NULL when called with the name of the last header.

Calling this function repeatedly returns all the header fields exactly once, but not necessarily in the order that they appear in the HTTP request.

The HTTP_HEADER function returns the value of the named HTTP header field, or NULL if not called from an HTTP service. It is used when processing an HTTP request via a web service. If a header for the given field name does not exist, the return value is NULL.

Here is a table of some typical HTTP headers and values.

Header Name	Header Value
Accept	image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
Accept-Language	en-us
UA-CPU	x86
Accept-Encoding	gzip, deflate
User-Agent	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.2; WOW64; SV1; .NET CLR 2.0.50727)
Host	localhost
Connection	Keep-Alive
@HttpMethod	GET
@HttpURI	/demo/ShowHTTPHeaders
@HttpVersion	HTTP/1.1

To get the value of each header, use the NEXT_HTTP_HEADER function to get the name of the header and then use the HTTP_HEADER function to get its value. The following example illustrates how to do this.

```
CREATE PROCEDURE HTTPHeaderExample()
RESULT ( html_string LONG VARCHAR )
BEGIN
    declare header_name long varchar;
    declare header_value long varchar;
```



```
    declare table_rows XML;
    set header_name  = NULL;
    set table_rows = NULL;
header_loop:
    LOOP
        SET header_name = NEXT_HTTP_HEADER( header_name );
        IF header_name IS NULL THEN
            LEAVE header_loop
        END IF;
        SET header_value = HTTP_HEADER( header_name );
        -- Format header name and value into an HTML table row
        SET table_rows = table_rows ||
            XMLELEMENT( name "tr",
                XMLATTRIBUTES( 'left' AS "align",
                               'top' AS "valign" ),
                XMLELEMENT( name "td", header_name ),
                XMLELEMENT( name "td", header_value ) );

    END LOOP;
    SELECT XMLELEMENT( name "table",
                XMLATTRIBUTES( '' AS "BORDER",
                              '10' AS "CELLPADDING",
                               '0' AS "CELLSPACING" ),
                XMLELEMENT( name "th",
                    XMLATTRIBUTES( 'left' AS "align",
                                   'top'  AS "valign" ),
                                   'Header Name' ),
                XMLELEMENT( name "th",
                    XMLATTRIBUTES( 'left' AS "align",
                                   'top'  AS "valign" ),
                                   'Header Value' ),
                table_rows );
END;
```

This example formats the header names and values into an HTML table. The following service can be defined to show how this sample procedure works.

```
CREATE SERVICE ShowHTTPHeaders
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS CALL HTTPHeaderExample();
```

To test the service, open a web browser and supply the following URL:

http://localhost:80/demo/ShowHTTPHeaders

To set the status code (or response code) of the request being processed, use the @HttpStatus special header. See “sa_set_http_header system procedure” [*SQL Anywhere Server - SQL Reference*].

For more information about header processing, see “Web services functions” [*SQL Anywhere Server - SQL Reference*].

Using SOAP services

To illustrate many of the features of web services, start with a simple Fahrenheit to Celsius temperature convertor as a sample service.

To set up a simple web service server

1. Create a database.

```
dbinit ftc
```

2. Start a server using this database.

```
dbsrv11 -xs http(port=8082) -n ftc ftc.db
```

3. Connect to the server using Interactive SQL.

```
dbisql -c "UID=DBA;PWD=sql;ENG=ftc"
```

4. Using Interactive SQL, create a web service.

```
CREATE SERVICE FtoCService
TYPE 'SOAP'
FORMAT 'XML'
AUTHORIZATION OFF
USER DBA
AS CALL FToCConvertor( :temperature );
```

5. Define the stored procedure that this service is to call to perform the calculation needed to convert from a temperature expressed in degrees Fahrenheit to a temperature expressed in degrees Celsius:

```
CREATE PROCEDURE FToCConvertor( temperature FLOAT )
BEGIN
    SELECT ROUND((temperature - 32.0) * 5.0 / 9.0, 5)
    AS answer;
END;
```

At this point, you now have a SQL Anywhere web service server running and ready to handle requests. The server is listening for SOAP requests on port 8082.

So how can you test this SOAP request server? The simplest way to do this is to use another SQL Anywhere database server to communicate the SOAP request and retrieve the response.

To send and receive SOAP requests

1. Create another database for use with a second server.

```
dbinit ftc_client
```

2. Start the personal server using this database.

```
dbeng11 ftc_client.db
```

3. Connect to the personal server using another instance of Interactive SQL.

```
dbisql -c "UID=DBA;PWD=sql;ENG=ftc_client"
```

4. Using Interactive SQL, create a stored procedure.


```
CREATE PROCEDURE FtoC( temperature FLOAT )
  URL 'http://localhost:8082/FtoCService'
  TYPE 'SOAP:DOC';
```

The URL clause is used to reference the SOAP web service. The string `'http://localhost:8082/FtoCService'` specifies the URI of the web service that is going to be used. This is a reference to the web server that is listening on port 8082.

The default format used when making a web service request is 'SOAP:RPC'. The format chosen in this example is 'SOAP:DOC', which is similar to 'SOAP:RPC' but allows for a richer set of data types. SOAP requests are always sent as XML documents. The mechanism for sending SOAP requests is 'HTTP:POST'.

5. You need a wrapper for the FtoC stored procedure, so create a second stored procedure.

```
CREATE PROCEDURE FahrenheitToCelsius( temperature FLOAT )
BEGIN
    DECLARE result LONG VARCHAR;
    DECLARE err INTEGER;
    DECLARE crsr CURSOR FOR
        CALL FtoC( temperature );

    OPEN crsr;
    FETCH crsr INTO result, err;
    CLOSE crsr;

    SELECT temperature, Celsius
    FROM OPENXML(result, '//tns:answer', 1, result)
         WITH ("Celsius" FLOAT 'text()');
END;
```

This stored procedure acts as a cover procedure for the call to the web service. The FtoC stored procedure returns a result set that this stored procedure processes. The result set is a single XML string that looks like the following.

```
<tns:rowset xmlns:tns="http://localhost/ftc/FtoCService">
 <tns:row>
  <tns:answer>100</tns:answer>
 </tns:row>
</tns:rowset>
```

The OPENXML function is used to parse the XML that is returned, extracting the value that is the temperature in degrees Celsius.

6. Call the stored procedure in order to send the request and obtain the response.

```
CALL FahrenheitToCelsius(212);
```

The Fahrenheit temperature and the Celsius equivalent appear.

temperature	Celsius
212	100

At this point, a simple web service running on a SQL Anywhere web server has been demonstrated. As you have seen, other SQL Anywhere servers can communicate with this web server. There has been little control over the content of the SOAP requests and responses that have traveled between these servers. In the next section, you will see how this simple web service can be extended by adding your own SOAP headers.

Note
The web service can be provided by the same database server, but must not reside in the same database as the client function. Attempting to access a web service in the same database results in the error `403 Forbidden`.

For information about SOAP header processing, see "Working with SOAP headers" on page 789.

Working with SOAP headers

In this section, the simple web service that was introduced in "Using SOAP services" on page 786 is extended to handle SOAP headers.

If you have followed the steps outlined in the previous section, you can skip steps 1 through 4 and go directly to step 5.

To create a web service server

1. Create a database.

   ```
   dbinit ftc
   ```

2. Start a server using this database.

   ```
   dbsrv11 -xs http(port=8082) -n ftc ftc.db
   ```

3. Connect to the server using Interactive SQL.

   ```
   dbisql -c "UID=DBA;PWD=sql;ENG=ftc"
   ```

4. Using Interactive SQL, create a web service.

   ```
   CREATE SERVICE FtoCService
   TYPE 'SOAP'
   FORMAT 'XML'
   AUTHORIZATION OFF
   USER DBA
   AS CALL FToCConvertor( :temperature );
   ```

5. Define the stored procedure that this service is to call to perform the calculation needed to convert from a temperature expressed in degrees Fahrenheit to a temperature expressed in degrees Celsius. Unlike the example in the previous section, this one includes additional statements to process a special SOAP header. If you have already worked through the example in the previous section, change the CREATE below to ALTER since you are now going to modify the stored procedure.

   ```
   CREATE PROCEDURE FToCConvertor( temperature FLOAT )
   BEGIN
     DECLARE hd_key LONG VARCHAR;
     DECLARE hd_entry LONG VARCHAR;
     DECLARE alias LONG VARCHAR;
     DECLARE first_name LONG VARCHAR;
     DECLARE last_name LONG VARCHAR;
     DECLARE xpath LONG VARCHAR;
     DECLARE authinfo LONG VARCHAR;
     DECLARE namespace LONG VARCHAR;
     DECLARE mustUnderstand LONG VARCHAR;
   header_loop:
     LOOP
       SET hd_key = NEXT_SOAP_HEADER( hd_key );
       IF hd_key IS NULL THEN
         -- no more header entries
         LEAVE header_loop;
       END IF;
       IF hd_key = 'Authentication' THEN
         SET hd_entry = SOAP_HEADER( hd_key );
         SET xpath = '/*:' || hd_key || '/*:userName';
         SET namespace = SOAP_HEADER( hd_key, 1,
                                      '@namespace' );
   ```

```
        SET mustUnderstand = SOAP_HEADER( hd_key, 1,
                                  'mustUnderstand' );
        BEGIN
          -- parse the XML returned in the SOAP header
          DECLARE crsr CURSOR FOR
            SELECT *
            FROM OPENXML( hd_entry, xpath )
              WITH ( alias LONG VARCHAR '@*:alias',
                first_name LONG VARCHAR '*:first/text()',
                last_name LONG VARCHAR '*:last/text()' );
          OPEN crsr;
          FETCH crsr INTO alias, first_name, last_name;
          CLOSE crsr;
        END;
        -- build a response header
        -- based on the pieces from the request header
        SET authinfo =
          XMLELEMENT( 'Authentication',
            XMLATTRIBUTES(
              namespace as xmlns,
              alias,
              mustUnderstand ),
              XMLELEMENT( 'first', first_name ),
              XMLELEMENT( 'last', last_name ) );
        CALL SA_SET_SOAP_HEADER( 'authinfo', authinfo );
      END IF;
    END LOOP header_loop;
    SELECT ROUND((temperature - 32.0) * 5.0 / 9.0, 5)
    AS answer;
  END;
```

Headers in SOAP requests can be obtained using a combination of the NEXT_SOAP_HEADER and SOAP_HEADER functions. The NEXT_SOAP_HEADER function iterates through the SOAP headers included within a request and returns the next SOAP header name. Calling it with NULL causes it to return the name of the first header. Subsequent headers are retrieved by passing the name of the previous header to the NEXT_SOAP_HEADER function. This function returns NULL when called with the name of the last header. The SQL code that does the SOAP header retrieval in the example is this. It exits the loop when NULL is finally returned.

```
SET hd_key = NEXT_SOAP_HEADER( hd_key );
    IF hd_key IS NULL THEN
      -- no more header entries
      LEAVE header_loop;
    END IF;
```

Calling this function repeatedly returns all the header fields exactly once, but not necessarily in the order they appear in the SOAP request.

The SOAP_HEADER function returns the value of the named SOAP header field, or NULL if not called from an SOAP service. It is used when processing an SOAP request via a web service. If a header for the given field-name does not exist, the return value is NULL.

The example searches for a SOAP header named Authentication. When it finds this header, it extracts the value for entire SOAP header as well as the values of the `@namespace` and `mustUnderstand` attributes. The SOAP header value might look something like this XML string:

```
<Authentication xmlns="SecretAgent" mustUnderstand="1">
  <userName alias="99">
    <first>Susan</first>
```



```
    <last>Hilton</last>
  </userName>
</Authentication>
```

For this header, the `@namespace` attribute value would be:`SecretAgent`

Also, the `mustUnderstand` attribute value would be:`1`

The interior of this XML string is parsed with the OPENXML function using an XPath string set to `/*:Authentication/*:userName`.

```
SELECT *
FROM OPENXML( hd_entry, xpath )
  WITH ( alias LONG VARCHAR '@*:alias',
  first_name LONG VARCHAR '*:first/text()',
  last_name LONG VARCHAR '*:last/text()' );
```

Using the sample SOAP header value shown above, the SELECT statement would create a result set as follows:

alias	first_name	last_name
99	Susan	Hilton

A cursor is declared on this result set and the three column values are fetched into three variables. At this point, you have all the information of interest that was passed to the web service. You have the temperature in Fahrenheit degrees and you have some additional attributes that were passed to the web service in a SOAP header. So what could you do with this information?

You could look up the name and alias that were provided to see if the person is authorized to use the web service. However, this exercise is not shown in the example.

The next step in the stored procedure is to create a response in the SOAP format. You can build the XML response as follows:

```
SET authinfo =
  XMLELEMENT( 'Authentication',
    XMLATTRIBUTES(
      namespace as xmlns,
      alias,
      mustUnderstand ),
      XMLELEMENT( 'first', first_name ),
      XMLELEMENT( 'last', last_name ) );
```

This builds the following XML string:

```
<Authentication xmlns="SecretAgent" alias="99"
                          mustUnderstand="1">
  <first>Susan</first>
  <last>Hilton</last>
</Authentication>
```

Finally, to return the SOAP response to the caller, the SA_SET_SOAP_HEADER stored procedure is used:

```
CALL SA_SET_SOAP_HEADER( 'authinfo', authinfo );
```

As in the example in the previous section, the last step is the calculation that converts from degrees Fahrenheit to degrees Celsius.

At this point, you now have a SQL Anywhere web service server running that can convert temperatures from degrees Fahrenheit to degrees Celsius as in the previous section. The major difference, however, is that it can also process a SOAP header from the caller and send a SOAP response back to the caller.

This is only half of the picture. The next step is to develop an example client that can send SOAP requests and receive SOAP responses.

If you have followed the steps outlined in the previous section, you can skip steps 1 through 3 and go directly to step 4.

To send and receive SOAP headers

1. Create another database for use with a second server.

   ```
   dbinit ftc_client
   ```

2. Start the personal server using this database.

   ```
   dbeng11 ftc_client.db
   ```

3. Connect to the personal server using another instance of Interactive SQL.

   ```
   dbisql -c "UID=DBA;PWD=sql;ENG=ftc_client"
   ```

4. Using Interactive SQL, create a stored procedure.

   ```
   CREATE PROCEDURE FtoC( temperature FLOAT,
       INOUT inoutheader LONG VARCHAR,
       IN inheader LONG VARCHAR )
     URL 'http://localhost:8082/FtoCService'
     TYPE 'SOAP:DOC'
     SOAPHEADER '!inoutheader!inheader';
   ```

   The URL clause is used to reference the SOAP web service. The string `'http://localhost:8082/FtoCService'` specifies the URI of the web service that is going to be used. This is a reference to the web server that is listening on port 8082.

   The default format used when making a web service request is 'SOAP:RPC'. The format chosen in this example is 'SOAP:DOC', which is similar to 'SOAP:RPC' but allows for a richer set of datatypes. SOAP requests are always sent as XML documents. The mechanism for sending SOAP requests is 'HTTP:POST'.

   The substitution variables in a SQL Anywhere client procedure (inoutheader, inheader) must be alpha-numeric. If the web service client is declared as a function, all its parameters are IN mode only (they cannot be assigned by the called function). Therefore, OPENXML or other string functions will have to be used to extract the SOAP response header information.

5. You need a wrapper for the FtoC stored procedure so create a second stored procedure as follows. Unlike the example in the previous section, this one includes additional statements to create a special SOAP header, send it in a web service call, and process a response from the web server. If you have already worked through the example in the previous section, change the CREATE below to ALTER since you are now going to modify the stored procedure.

   ```
   CREATE PROCEDURE FahrenheitToCelsius( temperature FLOAT )
   BEGIN
   ```



```
  DECLARE io_header LONG VARCHAR;
  DECLARE in_header LONG VARCHAR;
  DECLARE result LONG VARCHAR;
  DECLARE err INTEGER;
  DECLARE crsr CURSOR FOR
    CALL FtoC( temperature, io_header, in_header );
  SET io_header =
    '<Authentication xmlns="SecretAgent" ' ||
                    'mustUnderstand="1">' ||
    '<userName alias="99">' ||
    '<first>Susan</first><last>Hilton</last>' ||
    '</userName>' ||
    '</Authentication>';
  SET in_header =
    '<Session xmlns="SomeSession">' ||
    '123456789' ||
    '</Session>';

  MESSAGE 'send, soapheader=' || io_header || in_header;
  OPEN crsr;
  FETCH crsr INTO result, err;
  CLOSE crsr;
  MESSAGE 'receive, soapheader=' || io_header;
  SELECT temperature, Celsius
  FROM OPENXML(result, '//tns:answer', 1, result)
        WITH ("Celsius" FLOAT 'text()');
END;
```

This stored procedure acts as a cover procedure for the call to the web service. The stored procedure has been enhanced from the example in the previous section. It creates two SOAP headers. The first one is this.

```
<Authentication xmlns="SecretAgent"
                            mustUnderstand="1">
  <userName alias="99">
    <first>Susan</first>
    <last>Hilton</last>
  </userName></Authentication>
```

The second one is this.

```
<Session xmlns="SomeSession">123456789</Session>
```

When the cursor is opened, the SOAP request is sent to the web service.

```
<Authentication xmlns="SecretAgent" alias="99"
                            mustUnderstand="1">
<first>Susan</first>
<last>Hilton</last>
</Authentication>
```

The FtoC stored procedure returns a result set that this stored procedure will process. The result set will look something like this.

```
<tns:rowset xmlns:tns="http://localhost/ftc/FtoCService">
 <tns:row>
  <tns:answer>100</tns:answer>
 </tns:row>
</tns:rowset>
```

The OPENXML function is used to parse the XML that is returned, extracting the value that is the temperature in degrees Celsius.

6. Call the stored procedure in order to send the request and obtain the response:

```
CALL FahrenheitToCelsius(212);
```

The Fahrenheit temperature and the Celsius equivalent appears.

temperature	Celsius
212.0	100.0

A SQL Anywhere web service client can be declared as either a function or a procedure. A SQL Anywhere client function declaration effectively restricts all parameters to in mode only (parameters cannot be assigned by the called function). Calling a SQL Anywhere web service function will return the raw SOAP envelope response whereas a procedure returns a result set.

A SOAPHEADER clause has been added to the create/alter procedure/function statements. A SOAP header can be declared as a static constant or can be dynamically set using the parameter substitution mechanism. A web service client function can define one or more in mode substitution parameters whereas a web service client procedure can also define a single inout or out substitution parameter. Therefore a web service client procedure can return the response SOAP header within an out (or inout) substitution parameter. A web service function must parse the response SOAP envelope to obtain the header entries.

The following example illustrates how a client can specify the sending of several header entries with parameters and receiving the response SOAP header data.

```
CREATE PROCEDURE SoapClient(
    INOUT hd1 VARCHAR,
    IN hd2 VARCHAR,
    IN hd3 VARCHAR )
  URL 'localhost/some_endpoint'
  SOAPHEADER '!hd1!hd2!hd3';
```

Notes

- hd1, hd2, and hd3 all specify request header entries. hd1 also returns the aggregate of all response header entries.
- When the SOAP client is called with a SOAP header, it will generate the enclosing SOAP header element. If the SOAPHEADER value is NULL then no SOAP header element is generated.
- If no SOAP header is received, then hd1 is set to NULL.
- The INOUT mode specification of hd1 is redundant since the default mode of a parameter is INOUT.
- The following runtime error results if more than one substitution parameter is specified as an OUT (or INOUT) type: `'Expression has unsupported data type' SQLCODE=-624, ODBC 3 State-"HY000"`
- Only a single substitution parameter explicitly used for a SOAPHEADER can be declared as OUT.

Limitations

- Server side SOAP services cannot currently define input and output SOAP header requirements. Therefore SOAP header metadata is not available in the WSDL output of a DISH service. This means

that a SOAP client toolkit cannot automatically generate SOAP header interfaces for a SQL Anywhere SOAP service endpoint.

- Soap header faults are not supported.

Working with MIME types

The TYPE clause for a SQL Anywhere web service client procedure or function definition allows the specification of a MIME type. The value of the MIME type specification is used to set the Content-Type request header and set the mode of operation to allow only a single call parameter to populate the body of the request. Only zero or one parameter may remain when making a web service stored procedure (or function) call after parameter substitutions have been processed. Calling a web service procedure with a null or no parameter (after substitutions) will result in a request with no body and a content-length of zero. The behavior has not changed if a MIME type is not specified. Parameter names and values (multiple parameters are permitted) are URL encoded within the body of the HTTP request.

Some typical MIME types include:

- text/plain
- text/html
- text/xml

The following steps illustrate the setting of a MIME type. The first part sets up a web service that can be used to test the setting of MIME type. The second part demonstrates how to set a MIME type.

Create a web service server

1. Create a database.

   ```
   dbinit echo
   ```

2. Start a server using this database.

   ```
   dbsrv11 -xs http(port=8082) -n echo echo.db
   ```

3. Connect to the server using Interactive SQL.

   ```
   dbisql -c "UID=DBA;PWD=sql;ENG=echo"
   ```

4. Using Interactive SQL, create a web service.

   ```
   CREATE SERVICE EchoService
   TYPE 'RAW'
   USER DBA
   AUTHORIZATION OFF
   SECURE OFF
   AS CALL Echo(:valueAsXML);
   ```

5. Define the stored procedure that this service is to call.

   ```
   CREATE PROCEDURE Echo( parm LONG VARCHAR )
   BEGIN
       SELECT parm;
   END;
   ```

At this point, you now have a SQL Anywhere web service server running and ready to handle requests. The server is listening for HTTP requests on port 8082.

To use this web server for testing, create another SQL Anywhere database, start it, and connect to it. The following steps show how to do this.

To send an HTTP request

1. Using the database creation utility, create another database for use with a web service client.

   ```
   dbinit echo_client
   ```

2. Continuing with Interactive SQL, start this database using the following statement.

   ```
   START DATABASE 'echo_client.db'
   AS echo_client;
   ```

3. Now, connect to the database that has been started on the server echo using the following statement.

   ```
   CONNECT TO 'echo'
   DATABASE 'echo_client'
   USER 'DBA'
   IDENTIFIED BY 'sql';
   ```

4. Create a stored procedure that will communicate with the EchoService web service.

   ```
   CREATE PROCEDURE setMIME(
     value LONG VARCHAR,
     mimeType LONG VARCHAR,
     urlSpec LONG VARCHAR
     )
   URL '!urlSpec'
   HEADER 'ASA-Id'
   TYPE 'HTTP:POST:!mimeType';
   ```

   The URL clause is used to reference the web service. For illustration purposes, the URL will be passed as a parameter to the setMIME procedure.

   The TYPE clause indicates that the MIME type will be passed as a parameter to the setMIME procedure. The default format used when making a web service request is 'SOAP:RPC'. The format chosen for making this web service request is 'HTTP:POST'.

5. Call the stored procedure in order to send the request and obtain the response. The value parameter that is passed is a URL-encoded form of `<hello>this is xml</hello>`. The media type is `application/x-www-form-urlencoded` since form-urlencoded is understood by the SQL Anywhere web server. The URL for the web service is included as the final parameter in the call.

   ```
   CALL setMIME('valueAsXML=%3Chello%3Ethis%20is%20xml%3C/hello%3E',
       'application/x-www-form-urlencoded',
       'http://localhost:8082/EchoService');
   ```

   The final parameter specifies the URI of the web service that is listening on port 8082.

The following is representative of the HTTP packet that is sent to the web server.

```
POST /EchoService HTTP/1.0
Date: Sun, 28 Jan 2007 04:04:44 GMT
Host: localhost
Accept-Charset: windows-1252, UTF-8, *
User-Agent: SQLAnywhere/11.0.0.1297
Content-Type: application/x-www-form-urlencoded; charset=windows-1252
Content-Length: 49
ASA-Id: 1055532613:echo_client:echo:968000
Connection: close

valueAsXML=%3Chello%3Ethis%20is%20xml%3C/hello%3E
```

The following is the response from the web server.

```
HTTP/1.1 200 OK
Server: SQLAnywhere/11.0.0.1297
Date: Sun, 28 Jan 2007 04:04:44 GMT
Expires: Sun, 28 Jan 2007 04:04:44 GMT
Content-Type: text/plain; charset=windows-1252
Connection: close

<hello>this is xml</hello>
```

The result set that is displayed by Interactive SQL is shown next.

Attribute	Value
Status	HTTP/1.1 200 OK
Body	<hello>this is xml</hello>
Server	SQLAnywhere/11.0.0.1297
Date	Sun, 16 Dec 2007 04:04:44 GMT
Expires	Sun, 16 Dec 2007 04:04:44 GMT
Content-Type	text/plain; charset=windows-1252
Connection	close

Using HTTP sessions

HTTP connections can create an HTTP session to maintain state between HTTP requests.

An **HTTP session** provides the means for persisting the client (typically a web-browser) state with minimal SQL application code. A database connection under a session context is held for the duration of the session's lifetime. Each new HTTP request that is marked with a session ID is serialized (queued) so that each request with the same session ID is sequentially processed using the same database connection. Reusing the database connection provides the means of maintaining state information between HTTP requests. In contrast, sessionless HTTP requests create a new database connection for each request and data from temporary tables and connection variables cannot be shared across requests.

HTTP session management provides support for both URL and cookie state management techniques.

A working example of the HTTP session features is provided in *samples-dir\SQLAnywhere\HTTP \session.sql*.

Creating an HTTP session

A session is created within a web application using the HTTP option SessionID by calling the sa_set_http_option system procedure. A session ID can be any non-null string. Internally, a session key composed of the session ID and database name is generated so that the session key is unique across databases in the event that multiple databases are loaded. The entire session key is limited to 128 characters in length. In this example, a unique session ID is generated and passed to sa_set_http_option.

```
DECLARE session_id VARCHAR(64);
DECLARE tm TIMESTAMP;
SET tm=now(*);
SET session_id = 'session_' ||
    CONVERT( VARCHAR, SECONDS(tm)*1000+DATEPART(millisecond,tm));
CALL sa_set_http_option('SessionID', session_id);
```

A web application can obtain the session ID through the SessionID connection property. This property is an empty string if no session ID is defined for the connection (that is, the connection is sessionless).

```
DECLARE session_id VARCHAR(64);
SELECT CONNECTION_PROPERTY( 'SessionID' ) INTO session_id;
```

Once the session is created with the sa_set_http_option procedure, a localhost client can access the session with the specified session ID (for example, session_63315422814117) running within the database *dbname* running the service session_service with the following URL.

```
http://localhost/dbname/session_service?sessionid=session_63315422814117
```

If only one database is connected, then the database name can be omitted.

```
http://localhost/session_service?sessionid=session_63315422814117
```


Session management with cookies

Cookie state management is supported using the sa_set_http_header system procedure with 'Set-Cookie' as the field name. Utilizing cookies for state management negates the need to include the session ID within the URL. Instead, the client provides the session ID within its HTTP cookie header. The downside to using cookies for state management is that cookie support cannot be depended upon in an unregulated environment where clients may have disabled cookies. Consequently, a web application should support both URL and cookie session state management. A URL session ID, as described in the previous section, takes precedence in the event that a client provides both a URL and cookie session ID. It is the web application's responsibility to delete the SessionID cookie in the event that the session expires or that the session is explicitly deleted (for example, sa_set_http_option('SessionID', NULL)).

```
DECLARE session_id VARCHAR(64);
DECLARE tm TIMESTAMP;
SET tm=now(*);
SET session_id = 'session_' ||
    CONVERT( VARCHAR, SECONDS(tm)*1000+DATEPART(millisecond,tm));
CALL sa_set_http_option('SessionID', session_id);
CALL sa_set_http_header( 'Set-Cookie',
        'sessionid=' || session_id || ';' ||
        'max-age=60;' ||
        'path=/session;' );
```

Detection of stale sessions

The SessionID and SessionCreateTime connection properties are useful for determining whether the current connection is within a session context. If either connection property query returns an empty string, then the session does not exist. The SessionCreateTime property provides a metric of when a given session was created. It is defined immediately at the time that the sa_set_http_option call is made. The SessionLastTime property provides the time that the session was last used. More specifically, it is the time when the last processed request for the session released the database connection upon termination of that previous request. The SessionLastTime connection property is returned as an empty string when the session is first created until the request (that created the session) releases the connection.

Deleting or changing the session ID

The session ID can be reset to another value by calling the sa_set_http_option system procedure with a new SessionID value. Changing the session ID has the effect of deleting the old session and creating a new session, but it reuses the current database connection so that state information is not lost. A SessionID can be set to NULL (or the empty string) which deletes the session. A SessionID cannot be set to an ID of an existing session (other than its own session ID in which case nothing happens). Trying to set a SessionID to an existing session's session ID will result in an "Invalid setting for HTTP option 'SessionID' SQLCODE=-939" error.

A server receiving a burst of multiple HTTP requests specifying the same session context queues (serializes) the requests on its session queue. In the event that the SessionID is changed or deleted by one (or more) of the requests, any pending requests in the session queue are requeued as individual HTTP requests. Each HTTP request will fail to obtain the session because it no longer exists. An HTTP request failing to obtain a session will default to sessionless operation and create a new database connection. The web application

can verify that a request is operating within a session context by checking for a non-empty string value for the SessionID or SessionCreateTime connection properties. Of course, the web application can check the state of any application specific variables or temporary tables that it uses. Here is an example:

```
IF VAREXISTS( 'state' ) = 0 THEN
    // first invocation by this connection
    CREATE VARIABLE state LONG VARCHAR;
END IF;
```

Session semantics

An HTTP request can create an HTTP session context. A session created by the request is always immediately instantiated so that any subsequent HTTP requests requiring that session context is queued by the session.

An HTTP request that begins with no session but has created its session context is the creator of its session. The creator request can change or delete its session context where any change or deletion occurs immediately. An HTTP request that begins within a session context can also change or delete its session. A change made to its session immediately creates a pending session that is fully functional with the exception that another HTTP request cannot take ownership (an incoming request requiring the pending session would instead be queued on the session). To summarize, changing or deleting a session of a creator request modifies the current session context immediately, while a request only changing its session modifies its pending session. When an HTTP request finishes, it checks to see if it has a pending session. If a pending session exists, it deletes its current session and replaces it with the pending session. The database connection cached by the session is effectively moved to the new session context and all state data, such as temporary tables and created variables, are preserved.

In all cases, whenever an HTTP session is deleted, any requests within its queue are released and allowed to execute without a session context. Application code expecting that a request is running within a session context must attempt to acquire a valid session context by calling CONNECTION_PROPERTY('SessionID').

```
DECLARE ses_id LONG VARCHAR;
SELECT CONNECTION_PROPERTY( 'SessionID' ) INTO ses_id;
```

If an HTTP request is canceled either intentionally or because of network failure, an existing pending session is deleted preserving the original session context. A creator HTTP request, whether canceled or having terminated normally, changes session state immediately.

Dropping a connection and server shutdown

Explicitly dropping a database connection that is cached within a session context causes the session to be deleted. Deletion of the session in this manner is considered a cancel operation, and any HTTP requests released from the session queue are in a canceled state. This ensures that the HTTP requests are terminated quickly and signal the user appropriately.

Similarly, a server or database shutdown cancels its appropriate database connections possibly causing canceled HTTP requests.

Session timeout

The http_session_timeout public database option provides variable session timeout control. The option setting is in units of minutes. By default the public setting is 30 minutes. The minimum value is 1 minute and the maximum value is 525600 minutes (365 days). Web applications can change the timeout criteria from within any request that owns the session. A new timeout criteria may impact subsequent requests queued should the session timeout. It is up to the web application to provide the logic to detect if a client is attempting to access a non-existent session. It can do this by examining the SessionCreateTime connection property to determine if it is a valid timestamp. The SessionCreateTime value will be an empty string if the HTTP request is not associated with an existing session.

Session scope

Sessions are not persisted across server restarts.

Licensing

Since the connection associated with a session maintains its hold on a database connection for the life of the connection, it also holds one license seat as well. Web applications should take this into consideration and therefore ensure that stale sessions are deleted appropriately or have an appropriate timeout value set.

For more information about licensing in SQL Anywhere, visit http://www.sybase.com/detail?id=1056242.

Session errors

The error `503 Service Unavailable` occurs when a new request tries to access a session where more than 16 requests are pending on that session, or an error occurred while queuing the session.

The error `403 Forbidden` occurs if the client IP address or hostname does not match that of the creator of the session.

A request stipulating a session that does not exist does not implicitly generate an error. It is up to the web application to detect this condition (by checking SessionID, SessionCreateTime, or SessionLastTime connection properties) and do the appropriate action.

Summary of session connection properties and options

Connection properties

- **SessionID**

  ```
  SELECT CONNECTION_PROPERTY('SessionID') INTO ses_id;
  ```

  Provides the current session ID within the current database context.

- **SessionCreateTime**

  ```
  SELECT CONNECTION_PROPERTY('SessionCreateTime') INTO ses_create;
  ```

  Provides the timestamp of when the session was created.

- **SessionLastTime**

  ```
  SELECT CONNECTION_PROPERTY('SessionLastTime') INTO ses_last;
  ```

  Provides timestamp of when the session was released by the last request.

- **http_session_timeout**

  ```
  SELECT CONNECTION_PROPERTY('http_session_timeout') INTO ses_timeout;
  ```

  Fetches the current session timeout in units of minutes.

HTTP options

- **'SessionID','value'**

  ```
  CALL sa_set_http_option( 'SessionID', 'my_app_session_1' );
  ```

  Create or change a session context for the current HTTP request. Returns an error if my_app_session_1 is owned by another HTTP request.

- **'SessionID', NULL**

  ```
  CALL sa_set_http_option('SessionID', NULL );
  ```

  If the request comes from the session creator, the current session is deleted immediately; otherwise, the session is marked for deletion. Deleting a session when the request has no session is not an error and has no effect.

  Changing a session to a SessionID of the current session (has no pending session) is not an error and has no substantial effect.

  Changing a session to a SessionID in use by another HTTP request is an error.

  Changing a session when a change is already pending results in the pending session being deleted and new pending session being created.

  Changing a session with a pending session back to its original SessionID results in the pending session being deleted.

HTTP session timeout

- **http_session_timeout**

  ```
  SET TEMPORARY OPTION PUBLIC.http_session_timeout=100;
  ```

  Sets the current HTTP session timeout (in minutes). The default is 30 and the range is 1 to 525600 minutes (365 days).

  See “http_session_timeout option [database]” [*SQL Anywhere Server - Database Administration*].

Administration

A web application may require a means by which it can track active session usage within the database server. Session data can be found using the NEXT_CONNECTION function call to iterate through the active database connections and checking for session related properties such as SessionID. The following SQL code demonstrates this approach:

```
CREATE VARIABLE conn_id LONG VARCHAR;
CREATE VARIABLE the_sessionID LONG VARCHAR;
SELECT NEXT_CONNECTION( NULL, NULL ) INTO conn_id;
conn_loop:
LOOP
    IF conn_id IS NULL THEN
        LEAVE conn_loop;
    END IF;
    SELECT CONNECTION_PROPERTY( 'SessionID', conn_id )
        INTO the_sessionID;
    IF the_sessionID != '' THEN
        PRINT 'conn_id = %1!, SessionID = %2!', conn_id, the_sessionID;
    ELSE
        PRINT 'conn_id = %1!', conn_id;
    END IF;
    SELECT NEXT_CONNECTION( conn_id, NULL ) INTO conn_id;
END LOOP conn_loop;
PRINT '\n';
```

If you examine the database server messages window, you might see output similar to the following.

```
conn_id = 30
conn_id = 29, SessionID = session_63315442223323
conn_id = 28, SessionID = session_63315442220088
conn_id = 25, SessionID = session_63315441867629
```

Explicitly dropping a connection that belongs to a session causes the connection to be closed and the session to be deleted. If the connection being dropped is currently active in servicing an HTTP request, the request is marked for deletion and sent a cancel signal to terminate the request. When the request terminates, the session is deleted and the connection closed. Deleting the session causes any pending requests on that session's queue to be requeued as discussed in “Deleting or changing the session ID” on page 800. In the event the connection is currently inactive, the session is marked for deletion and requeued to the beginning of the session timeout queue. The session and the connection are deleted in the next timeout cycle (normally within 5 seconds). Any session marked for deletion cannot be used by a new HTTP request.

When stopping a database unconditionally, each database connection is dropped, causing all sessions under that database context (see the description of session key in “Creating an HTTP session” on page 799) to be deleted. This is guaranteed since one valid database connection must exist for one session context and a database connection can only be associated with one session at a time. Both the session and database connection must be within the same database context.

Using automatic character set conversion

By default, character set conversion is performed automatically on outgoing result sets of type text. Result sets of other types, such as binary objects, are not translated. The character set of the request is converted to the database character set, and the result set is converted from the database character set to the client character set, as required, except on binary columns in the result set. When the request lists multiple character sets that it can handle, the server takes the first suitable one from the list.

Character-set conversion can be enabled or disabled by setting the HTTP option CharsetConversion. The allowed values are ON and OFF. The default value is ON. The following statement turns automatic character-set conversion off:

```
CALL sa_set_http_option( 'CharsetConversion', 'OFF' );
```

For more information about built-in stored procedures, see "Alphabetical list of system procedures" [*SQL Anywhere Server - SQL Reference*].

Handling errors

When web service requests fail, the database server generates standard errors that appear in your browser. These errors are assigned numbers consistent with the protocol standards.

If the service is a SOAP service, faults are returned to the client as SOAP faults as defined in the SOAP version 1.1 standard:

- When an error in the application handling the request generates a SQLCODE, a SOAP Fault is returned with a faultcode of Client, possibly with a sub-category, such as Procedure. The faultstring element within the SOAP Fault is set to a detailed explanation of the error and a detail element contains the numeric SQLCODE value.
- In the event of a transport protocol error, the faultcode is set to either Client or Server, depending on the error, faultstring is set to the HTTP transport message, such as `404 Not Found`, and the detail element contains the numeric HTTP error value.
- SOAP Fault messages generated due to application errors that return a SQLCODE value are returned with an HTTP status of `200 OK`.

If the client cannot be identified as a SOAP client, then the appropriate HTTP error is returned in a generated HTML document.

The following are some of the typical errors that you may encounter:

Number	Name	SOAP fault	Description
301	Moved permanently	Server	The requested page has been permanently moved. The server automatically redirects the request to the new location.
304	Not Modified	Server	The server has decided, based on information in the request, that the requested data has not been modified since the last request and so it does not need to be sent again.
307	Temporary Redirect	Server	The requested page has been moved, but this change may not be permanent. The server automatically redirects the request to the new location.
400	Bad Request	Client.BadRequest	The HTTP request is incomplete or malformed.
401	Authorization Required	Client.Authorization	Authorization is required to use the service, but a valid user name and password were not supplied.
403	Forbidden	Client.Forbidden	You do not have permission to access the database.

Number	Name	SOAP fault	Description
404	Not Found	Client.NotFound	The named database is not running on the server, or the named web service does not exist.
408	Request Timeout	Server.RequestTimeout	The maximum connection idle time was exceeded while receiving the request.
411	HTTP Length Required	Client.LengthRequired	The server requires that the client include a Content-Length specification in the request. This typically occurs when uploading data to the server.
413	Entity Too Large	Server	The request exceeds the maximum permitted size.
414	URI Too Large	Server	The length of the URI exceeds the maximum allowed length.
500	Internal Server Error	Server	An internal error occurred. The request could not be processed.
501	Not Implemented	Server	The HTTP request method is not GET, HEAD, or POST.
502	Bad Gateway	Server	The document requested resides on a third-party server and the server received an error from the third-party server.
503	Service Unavailable	Server	The number of connections exceeds the allowed maximum.

Part IV. SQL Anywhere Database Tools Interface

This part describes the database tools programming interface for SQL Anywhere.

CHAPTER 22

Database tools interface

Contents

Introduction to the database tools interface

SQL Anywhere includes Sybase Central and a set of utilities for managing databases. These database management utilities perform tasks such as backing up databases, creating databases, translating transaction logs to SQL, and so on.

Supported platforms

All the database management utilities use a shared library called the **database tools library**. It is supplied for Windows operating systems and for Unix. The name of this library is *dbtool11.dll* for Windows, and *libdbtool11.so* (non-threaded) or *libdbtool11_r.so* (threaded) for Unix.

You can develop your own database management utilities or incorporate database management features into your applications by calling the database tools library. This chapter describes the interface to the database tools library. This chapter assumes you are familiar with how to call library routines from the development environment you are using.

The database tools library has functions, or entry points, for each of the database management utilities. In addition, functions must be called before use of other database tools functions and when you have finished using other database tools functions.

Windows Mobile

The *dbtool11.dll* library is supplied for Windows Mobile, but includes only entry points for DBToolsInit, DBToolsFini, DBRemoteSQL, and DBSynchronizeLog. Other entry points are not provided for Windows Mobile.

The dbtools.h header file

The dbtools header file included with SQL Anywhere lists the entry points to the DBTools library and also the structures used to pass information to and from the library. The *dbtools.h* file is installed into the *SDK \Include* subdirectory under your SQL Anywhere installation directory. You should consult the *dbtools.h* file for the latest information about the entry points and structure members.

The *dbtools.h* header file includes other files such as:

- **sqlca.h** This is included for resolution of various macros, not for the SQLCA itself.
- **dllapi.h** Defines preprocessor macros for operating-system dependent and language-dependent macros.
- **dbtlvers.h** Defines the DB_TOOLS_VERSION_NUMBER preprocessor macro and other version specific macros.

The sqldef.h header file

The *sqldef.h* header file includes error return values.

The dbrmt.h header file

The *dbrmt.h* header file included with SQL Anywhere describes the DBRemoteSQL entry point in the DBTools library and also the structure used to pass information to and from the DBRemoteSQL entry point. The *dbrmt.h* file is installed into the *SDK\Include* subdirectory under your SQL Anywhere installation

directory. You should consult the *dbrmt.h* file for the latest information about the DBRemoteSQL entry point and structure members.

Using the database tools interface

This section provides an overview of how to develop applications that use the DBTools interface for managing databases.

Using the import libraries

To use the DBTools functions, you must link your application against a DBTools **import library** that contains the required function definitions.

For Unix systems, no import library is required.

Import libraries

Import libraries for the DBTools interface are provided with SQL Anywhere for Windows and Windows Mobile. For Windows, they can be found in the *SDK\Lib\x86* and *SDK\Lib\x64* subdirectories under your SQL Anywhere installation directory. For Windows Mobile, the import library can be found in the *SDK\Lib\CE\Arm.50* subdirectory under your SQL Anywhere installation directory. The provided DBTools import libraries are as follows:

Compiler	Library
Borland (32-bit only)	*dbtlstb.lib*
Microsoft Windows	*dbtlstm.lib*
Microsoft Windows Mobile	*dbtool11.lib*
Watcom (32-bit only)	*dbtlstw.lib*

Starting and finishing the DBTools library

Before using any other DBTools functions, you must call DBToolsInit. When you are finished using the DBTools library, you must call DBToolsFini.

The primary purpose of the DBToolsInit and DBToolsFini functions is to allow the DBTools library to load the SQL Anywhere message library. The messages library contains localized versions of all error messages and prompts that DBTools uses internally. If DBToolsFini is not called, the reference count of the messages library is not decremented and it will not be unloaded, so be careful to ensure there is a matched pair of DBToolsInit/DBToolsFini calls.

The following code fragment illustrates how to initialize and clean up DBTools:

```
// Declarations
a_dbtools_info  info;
short           ret;

//Initialize the a_dbtools_info structure
```



```
memset( &info, 0, sizeof( a_dbtools_info) );
info.errorrtn = (MSG_CALLBACK)MyErrorRtn;

// initialize the DBTools library
ret = DBToolsInit( &info );
if( ret != EXIT_OKAY ) {
    // library initialization failed
    ...
}
// call some DBTools routines ...
...
// finalize the DBTools library
DBToolsFini( &info );
```

Calling the DBTools functions

All the tools are run by first filling out a structure, and then calling a function (or **entry point**) in the DBTools library. Each entry point takes a pointer to a single structure as argument.

The following example shows how to use the DBBackup function on a Windows operating system.

```
// Initialize the structure
a_backup_db backup_info;
memset( &backup_info, 0, sizeof( backup_info ) );

// Fill out the structure
backup_info.version = DB_TOOLS_VERSION_NUMBER;
backup_info.output_dir = "c:\\backup";
backup_info.connectparms ="UID=DBA;PWD=sql;DBF=demo.db";

backup_info.confirmrtn = (MSG_CALLBACK) ConfirmRtn ;
backup_info.errorrtn = (MSG_CALLBACK) ErrorRtn ;
backup_info.msgrtn = (MSG_CALLBACK) MessageRtn ;
backup_info.statusrtn = (MSG_CALLBACK) StatusRtn ;
backup_info.backup_database = TRUE;

// start the backup
DBBackup( &backup_info );
```

For information about the members of the DBTools structures, see "DBTools structures" on page 831.

Using callback functions

Several elements in DBTools structures are of type MSG_CALLBACK. These are pointers to callback functions.

Uses of callback functions

Callback functions allow DBTools functions to return control of operation to the user's calling application. The DBTools library uses callback functions to handle messages sent to the user by the DBTools functions for four purposes:

- **Confirmation** Called when an action needs to be confirmed by the user. For example, if the backup directory does not exist, the tools library asks if it needs to be created.

- **Error message** Called to handle a message when an error occurs, such as when an operation is out of disk space.
- **Information message** Called for the tools to display some message to the user (such as the name of the current table being unloaded).
- **Status information** Called for the tools to display the status of an operation (such as the percentage done when unloading a table).

Assigning a callback function to a structure

You can directly assign a callback routine to the structure. The following statement is an example using a backup structure:

```
backup_info.errorrtn = (MSG_CALLBACK) MyFunction
```

MSG_CALLBACK is defined in the *dllapi.h* header file supplied with SQL Anywhere. Tools routines can call back to the calling application with messages that should appear in the appropriate user interface, whether that be a windowing environment, standard output on a character-based system, or other user interface.

Confirmation callback function example

The following example confirmation routine asks the user to answer YES or NO to a prompt and returns the user's selection:

```
extern short _callback ConfirmRtn(
        char * question )
{
    int ret = IDNO;
    if( question != NULL ) {
        ret = MessageBox( HwndParent, question,
        "Confirm", MB_ICONEXCLAMATION|MB_YESNO );
    }
    return( ret == IDYES );
}
```

Error callback function example

The following is an example of an error message handling routine, which displays the error message in a window.

```
extern short _callback ErrorRtn(
        char * errorstr )
{
    if( errorstr != NULL ) {
        MessageBox( HwndParent, errorstr, "Backup Error", MB_ICONSTOP|
MB_OK );
    }
    return( 0 );
}
```

Message callback function example

A common implementation of a message callback function outputs the message to the screen:

```
extern short _callback MessageRtn(
        char * messagestr )
{
    if( messagestr != NULL ) {
                    OutputMessageToWindow( messagestr );
```



```
    }
    return( 0 );
}
```

Status callback function example

A status callback routine is called when a tool needs to display the status of an operation (like the percentage done unloading a table). A common implementation would just output the message to the screen:

```
extern short _callback StatusRtn(
        char * statusstr )
{
    if( statusstr != NULL ) {
        OutputMessageToWindow( statusstr );
    return( 0 );
}
```

Version numbers and compatibility

Each structure has a member that indicates the version number. You should use this version member to hold the version of the DBTools library that your application was developed against. The current version of the DBTools library is defined when you include the *dbtools.h* header file.

To assign the current version number to a structure

- Assign the version constant to the version member of the structure before calling the DBTools function. The following line assigns the current version to a backup structure:

  ```
  backup_info.version = DB_TOOLS_VERSION_NUMBER;
  ```

Compatibility

The version number allows your application to continue working against newer versions of the DBTools library. The DBTools functions use the version number supplied by your application to allow the application to work, even if new members have been added to the DBTools structure.

When any of the DBTools structures are updated, or when a newer version of the software is released, the version number is augmented. If you use DB_TOOLS_VERSION_NUMBER and you rebuild your application with a new version of the DBTools header file, then you must deploy a new version of the DBTools library. If the functionality of your application doesn't change, then you may want to use one of the version-specific macros defined in *dbtlsvers.h*, so that a library version mismatch does not occur.

Using bit fields

Many of the DBTools structures use bit fields to hold Boolean information in a compact manner. For example, the backup structure includes the following bit fields:

```
a_bit_field     backup_database : 1;
a_bit_field     backup_logfile  : 1;
a_bit_field     no_confirm      : 1;
a_bit_field     quiet           : 1;
a_bit_field     rename_log      : 1;
a_bit_field     truncate_log    : 1;
```

```
a_bit_field     rename_local_log: 1;
a_bit_field     server_backup   : 1;
```

Each bit field is one bit long, indicated by the 1 to the right of the colon in the structure declaration. The specific data type used depends on the value assigned to a_bit_field, which is set at the top of *dbtools.h*, and is operating system-dependent.

You assign a value of 0 or 1 to a bit field to pass Boolean information in the structure.

A DBTools example

You can find this sample and instructions for compiling it in the *samples-dir\SQLAnywhere\DBTools* directory. The sample program itself is in *main.cpp*. The sample illustrates how to use the DBTools library to perform a backup of a database.

```
#define WIN32

#include <stdio.h>
#include <string.h>
#include "windows.h"
#include "sqldef.h"
#include "dbtools.h"
extern short _callback ConfirmCallBack( char * str )
{
    if( MessageBox( NULL, str, "Backup",
         MB_YESNO|MB_ICONQUESTION ) == IDYES )
    {
        return 1;
    }
    return 0;
}
extern short _callback MessageCallBack( char * str )
{
    if( str != NULL )
    {
        fprintf( stdout, "%s\n", str );
    }
    return 0;
}
extern short _callback StatusCallBack( char * str )
{
    if( str != NULL )
    {
        fprintf( stdout, "%s\n", str );
    }
    return 0;
}
extern short _callback ErrorCallBack( char * str )
{
    if( str != NULL )
    {
        fprintf( stdout, "%s\n", str );
    }
    return 0;
}
typedef void (CALLBACK *DBTOOLSPROC)( void * );
typedef short (CALLBACK *DBTOOLSFUNC)( void * );

// Main entry point into the program.
```



```
int main( int argc, char * argv[] )
{
    a_dbtools_info  dbt_info;
    a_backup_db     backup_info;
    char            dir_name[ _MAX_PATH + 1];
    char            connect[ 256 ];
    HINSTANCE       hinst;
    DBTOOLSFUNC     dbbackup;
    DBTOOLSFUNC     dbtoolsinit;
    DBTOOLSPROC     dbtoolsfini;
    short           ret_code;

    // Always initialize to 0 so new versions
    // of the structure will be compatible.
    memset( &dbt_info, 0, sizeof( a_dbtools_info ) );
    dbt_info.errorrtn = (MSG_CALLBACK)MessageCallBack;;

    memset( &backup_info, 0, sizeof( a_backup_db ) );
    backup_info.version = DB_TOOLS_VERSION_NUMBER;
    backup_info.quiet = 0;
    backup_info.no_confirm = 0;
    backup_info.confirmrtn = (MSG_CALLBACK)ConfirmCallBack;
    backup_info.errorrtn = (MSG_CALLBACK)ErrorCallBack;
    backup_info.msgrtn = (MSG_CALLBACK)MessageCallBack;
    backup_info.statusrtn = (MSG_CALLBACK)StatusCallBack;
    if( argc > 1 )
    {
        strncpy( dir_name, argv[1], _MAX_PATH );
    }
    else
    {
        // DBTools does not expect (or like) a trailing slash
        strcpy( dir_name, "c:\\temp" );
    }
    backup_info.output_dir = dir_name;
    if( argc > 2 )
    {
        strncpy( connect, argv[2], 255 );
    }
    else
    {
        strcpy( connect, "DSN=SQL Anywhere 11 Demo" );
    }
    backup_info.connectparms = connect;
    backup_info.quiet = 0;
    backup_info.no_confirm = 0;
    backup_info.backup_database = 1;
    backup_info.backup_logfile = 1;
    backup_info.rename_log = 0;
    backup_info.truncate_log = 0;
    hinst = LoadLibrary( "dbtool11.dll" );
    if( hinst == NULL )
    {
        // Failed
        return EXIT_FAIL;
    }
    dbbackup = (DBTOOLSFUNC) GetProcAddress( (HMODULE)hinst,
        "_DBBackup@4" );
    dbtoolsinit = (DBTOOLSFUNC) GetProcAddress( (HMODULE)hinst,
        "_DBToolsInit@4" );
    dbtoolsfini = (DBTOOLSPROC) GetProcAddress( (HMODULE)hinst,
        "_DBToolsFini@4" );
    ret_code = (*dbtoolsinit)( &dbt_info );
    if( ret_code != EXIT_OKAY ) {
```

```
            return ret_code;
        }
        ret_code = (*dbbackup)( &backup_info );
        (*dbtoolsfini)( &dbt_info );
        FreeLibrary( hinst );
        return ret_code;
    }
```


DBTools functions

DBBackup function

Backs up a database. This function is used by the dbbackup utility.

Prototype

short **DBBackup (** const a_backup_db * **);**

Parameters

A pointer to a structure. See "a_backup_db structure" on page 831.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

The DBBackup function manages all client-side database backup tasks.

For a description of these tasks, see "Backup utility (dbbackup)" [*SQL Anywhere Server - Database Administration*].

To perform a server-side backup, use the BACKUP DATABASE statement. See "BACKUP statement" [*SQL Anywhere Server - SQL Reference*].

DBChangeLogName function

Changes the name of the transaction log file. This function is used by the dblog utility.

Prototype

short **DBChangeLogName (** const a_change_log * **);**

Parameters

A pointer to a structure. See "a_change_log structure" on page 833.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

The -t option of the Transaction Log utility (dblog) changes the name of the transaction log. DBChangeLogName provides a programmatic interface to this function.

For a description of the dblog utility, see "Transaction Log utility (dblog)" [*SQL Anywhere Server - Database Administration*].

See also

- "ALTER DATABASE statement" [*SQL Anywhere Server - SQL Reference*]

DBCreate function

Creates a database. This function is used by the dbinit utility.

Prototype

short **DBCreate (** const a_create_db * **);**

Parameters

A pointer to a structure. See "a_create_db structure" on page 835.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the dbinit utility, see "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*].

See also

- "CREATE DATABASE statement" [*SQL Anywhere Server - SQL Reference*]

DBCreatedVersion function

Determines the version of SQL Anywhere that was used to create a database file, without attempting to start the database. Currently, this function only differentiates between version 10 or 11 and pre-10 databases.

Prototype

short **DBCreatedVersion (** a_db_version_info * **);**

Parameters

A pointer to a structure. See "a_db_version_info structure" on page 839.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

If the return code indicates success, then the created_version field of the a_db_version_info structure contains a value of type a_db_version indicating which version of SQL Anywhere created the database. For the definition of the possible values, see "a_db_version enumeration" on page 872.

Version information is not set if a failing code is returned.

See also

- "CREATE DATABASE statement" [*SQL Anywhere Server - SQL Reference*]
- "a_db_version_info structure" on page 839
- "a_db_version enumeration" on page 872

DBErase function

Erases a database file and/or transaction log file. This function is used by the dberase utility.

Prototype

short **DBErase (** const an_erase_db * **);**

Parameters

A pointer to a structure. See "an_erase_db structure" on page 842.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the Erase utility and its features, see "Erase utility (dberase)" [*SQL Anywhere Server - Database Administration*].

DBInfo function

Returns information about a database file. This function is used by the dbinfo utility.

Prototype

short **DBInfo (** const a_db_info * **);**

Parameters

A pointer to a structure. See "a_db_info structure" on page 837.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the Information utility and its features, see "Information utility (dbinfo)" [*SQL Anywhere Server - Database Administration*].

See also

- "DBInfoDump function" on page 824
- "DBInfoFree function" on page 824
- "DB_PROPERTY function [System]" [*SQL Anywhere Server - SQL Reference*]

DBInfoDump function

Returns information about a database file. This function is used by the dbinfo utility when the -u option is used.

Prototype

short **DBInfoDump (** const a_db_info * **);**

Parameters

A pointer to a structure. See "a_db_info structure" on page 837.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the Information utility and its features, see "Information utility (dbinfo)" [*SQL Anywhere Server - Database Administration*].

See also

- "DBInfo function" on page 823
- "DBInfoFree function" on page 824
- "sa_table_page_usage system procedure" [*SQL Anywhere Server - SQL Reference*]

DBInfoFree function

Frees resources after the DBInfoDump function is called.

Prototype

short **DBInfoFree (** const a_db_info * **);**

Parameters

A pointer to a structure. See "a_db_info structure" on page 837.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the Information utility and its features, see "Information utility (dbinfo)" [*SQL Anywhere Server - Database Administration*].

See also

- "DBInfo function" on page 823
- "DBInfoDump function" on page 824

DBLicense function

Modifies or reports the licensing information of the database server.

Prototype

short **DBLicense (** const a_db_lic_info * **);**

Parameters

A pointer to a structure. See "a_dblic_info structure" on page 840.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the Server Licensing utility and its features, see "Server Licensing utility (dblic)" [*SQL Anywhere Server - Database Administration*].

DBRemoteSQL function

Accesses the SQL Remote Message Agent.

Prototype

short **DBRemoteSQL(** const a _remote_sql * **);**

Parameters

A pointer to a structure. See "a_remote_sql structure" on page 843.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the features you can access, see "Message Agent" [*SQL Remote*].

See also

- "SQL Remote concepts" [*SQL Remote*]

DBSynchronizeLog function

Synchronize a database with a MobiLink server.

Prototype

short **DBSynchronizeLog(** const a _sync_db * **);**

Parameters

A pointer to a structure. See "a_sync_db structure" on page 849.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the features you can access, see "Initiating synchronization" [*MobiLink - Client Administration*].

See also

- "DBTools interface for dbmlsync" [*MobiLink - Client Administration*]

DBToolsFini function

Decrements the counter and frees resources when an application is finished with the DBTools library.

Prototype

short **DBToolsFini (** const a_dbtools_info * **);**

Parameters

A pointer to a structure. See "a_dbtools_info structure" on page 841.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

The DBToolsFini function must be called at the end of any application that uses the DBTools interface. Failure to do so can lead to lost memory resources.

See also

- "DBToolsInit function" on page 826

DBToolsInit function

Prepares the DBTools library for use.

Prototype

short **DBToolsInit(** const a_dbtools_info * **);**

Parameters

A pointer to a structure. See "a_dbtools_info structure" on page 841.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

The primary purpose of the DBToolsInit function is to load the SQL Anywhere messages library. The messages library contains localized versions of error messages and prompts that DBTools uses internally.

The DBToolsInit function must be called at the start of any application that uses the DBTools interface, before any other DBTools functions. For an example, see "A DBTools example" on page 818.

See also

- "DBToolsFini function" on page 826

DBToolsVersion function

Returns the version number of the DBTools library.

Prototype

short **DBToolsVersion (** void **);**

Return value

A short integer indicating the version number of the DBTools library.

Remarks

Use the DBToolsVersion function to check that the DBTools library is not older than one against which your application is developed. While applications can run against newer versions of DBTools, they cannot run against older versions.

See also

- "Version numbers and compatibility" on page 817

DBTranslateLog function

Translates a transaction log file to SQL. This function is used by the dbtran utility.

Prototype

short **DBTranslateLog (** const a_translate_log * **);**

Parameters

A pointer to a structure. See "a_translate_log structure" on page 858.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the Log Translation utility, see "Log Translation utility (dbtran)" [*SQL Anywhere Server - Database Administration*].

DBTruncateLog function

Truncates a transaction log file. This function is used by the dbbackup utility.

Prototype

short **DBTruncateLog (** const a_truncate_log * **);**

Parameters

A pointer to a structure. See "a_truncate_log structure" on page 862.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the Backup utility, see "Backup utility (dbbackup)" [*SQL Anywhere Server - Database Administration*].

See also

- "BACKUP statement" [*SQL Anywhere Server - SQL Reference*]

DBUnload function

Unloads a database. This function is used by the dbunload and dbxtract utilities.

Prototype

short **DBUnload (** const an_unload_db * **);**

Parameters

A pointer to a structure. See "an_unload_db structure" on page 863.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the Unload utility, see "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*].

For information about the Extraction utility, see "Extraction utility" [*SQL Remote*].

DBUpgrade function

Upgrades a database file. This function is used by the dbupgrad utility.

Prototype

short **DBUpgrade (** const an_upgrade_db * **);**

Parameters

A pointer to a structure. See "an_upgrade_db structure" on page 868.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the Upgrade utility, see "Upgrade utility (dbupgrad)" [*SQL Anywhere Server - Database Administration*].

See also

- "ALTER DATABASE statement" [*SQL Anywhere Server - SQL Reference*]

DBValidate function

Validates all or part of a database. This function is used by the dbvalid utility.

Prototype

short **DBValidate (** const a_validate_db * **);**

Parameters

A pointer to a structure. See "a_validate_db structure" on page 869.

Return value

A return code, as listed in "Software component exit codes" on page 878.

Remarks

For information about the Validation utility, see "Validation utility (dbvalid)" [*SQL Anywhere Server - Database Administration*].

> **Caution**
> Validating a table or an entire database should be performed while no connections are making changes to the database; otherwise, spurious errors may be reported indicating some form of database corruption even though no corruption actually exists.

See also

- "VALIDATE statement" [*SQL Anywhere Server - SQL Reference*]

- “sa_validate system procedure” [*SQL Anywhere Server - SQL Reference*]

DBTools structures

This section lists the structures that are used to exchange information with the DBTools library. The structures are listed alphabetically. With the exception of the a_remote_sql structure, all of these structures are defined in *dbtools.h*. The a_remote_sql structure is defined in *dbrmt.h*.

Many of the structure elements correspond to command line options on the corresponding utility. For example, several structures have a member named quiet, which can take on values of 0 or 1. This member corresponds to the quiet operation (-q) option used by many of the utilities.

a_backup_db structure

Holds the information needed to perform backup tasks using the DBTools library.

Syntax

```
typedef struct a_backup_db {
    unsigned short      version;
    const char *        output_dir;
    const char *        connectparms;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         backup_database : 1;
    a_bit_field         backup_logfile  : 1;
    a_bit_field         no_confirm      : 1;
    a_bit_field         quiet           : 1;
    a_bit_field         rename_log      : 1;
    a_bit_field         truncate_log    : 1;
    a_bit_field         rename_local_log: 1;
    a_bit_field         server_backup   : 1;
    const char *        hotlog_filename;
    char                backup_interrupted;
    a_chkpt_log_type    chkpt_log_type;
    a_sql_uint32        page_blocksize;
} a_backup_db;
```

Members

Member	Description
Version	DBTools version number.
output_dir	Path to the output directory. For example: `"c:\backup"`

Member	Description

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	The database server would be started by the connection string START parameter. For example:
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	A full example connection string including the START parameter:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

confirmrtn	Callback routine for confirming an action.
errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
statusrtn	Callback routine for handling a status message.
backup_database	Back up the database file (1) or not (0).
backup_logfile	Back up the transaction log file (1) or not (0).
no_confirm	Operate with (0) or without (1) confirmation.
quiet	Operate without printing messages (1), or print messages (0).
rename_log	Rename the transaction log.
truncate_log	Delete the transaction log.
rename_local_log	Rename the local backup of the transaction log.
server_backup	When set to 1, indicates backup on server using BACKUP DATABASE. Equivalent to dbbackup -s option.
hotlog_filename	File name for the live backup file.
backup_interrupted	Indicates that the operation was interrupted.

Member	Description
chkpt_log_type	Control copying of checkpoint log. Must be one of BACKUP_CHKPT_LOG_COPY, BACKUP_CHKPT_LOG_NOCOPY, BACKUP_CHKPT_LOG_RECOVER,BACKUP_CHKPT_LOG_AUTO, or BACKUP_CHKPT_LOG_DEFAULT.
page_blocksize	Number of pages in data blocks. Equivalent to dbbackup -b option. If set to 0, then the default is 128.

See also

- "DBBackup function" on page 821
- "a_db_version enumeration" on page 872
- "Using callback functions" on page 815

a_change_log structure

Holds the information needed to perform dblog tasks using the DBTools library.

Syntax

```
typedef struct a_change_log {
    unsigned short      version;
    const char *        dbname;
    const char *        logname;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         query_only              : 1;
    a_bit_field         quiet                   : 1;
    a_bit_field         change_mirrorname       : 1;
    a_bit_field         change_logname          : 1;
    a_bit_field         ignore_ltm_trunc        : 1;
    a_bit_field         ignore_remote_trunc     : 1;
    a_bit_field         set_generation_number   : 1;
    a_bit_field         ignore_dbsync_trunc     : 1;
    const char *        mirrorname;
    unsigned short      generation_number;
    char *              zap_current_offset;
    char *              zap_starting_offset;
    char *              encryption_key;
} a_change_log;
```

Members

Member	Description
version	DBTools version number.
dbname	Database file name.
logname	The name of the transaction log. If set to NULL, there is no log.

Member	Description
errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
query_only	If 1, just display the name of the transaction log. If 0, permit changing of the log name.
quiet	Operate without printing messages (1), or print messages (0).
change_mirrorname	If 1, permit changing of the log mirror name.
change_logname	If 1, permit changing of the transaction log name.
ignore_ltm_trunc	When using the Log Transfer Manager, performs the same function as the dbcc settrunc('ltm', 'gen_id', n) Replication Server function. For information about dbcc, see your Replication Server documentation.
ignore_remote_trunc	For SQL Remote. Resets the offset kept for the purposes of the delete_old_logs option, allowing transaction logs to be deleted when they are no longer needed.
set_generation_number	When using the Log Transfer Manager, used after a backup is restored to set the generation number.
ignore_dbsync_trunc	When using dbmlsync, resets the offset kept for the purposes of the delete_old_logs option, allowing transaction logs to be deleted when they are no longer needed.
mirrorname	The new name of the transaction log mirror file.
generation_number	The new generation number. Used together with set_generation_number.
zap_current_offset	Change the current offset to the specified value. This is for use only in resetting a transaction log after an unload and reload to match dbremote or dbmlsync settings.
zap_starting_offset	Change the starting offset to the specified value. This is for use only in resetting a transaction log after an unload and reload to match dbremote or dbmlsync settings.
encryption_key	The encryption key for the database file.

See also

- "DBChangeLogName function" on page 821

- "Using callback functions" on page 815

a_create_db structure

Holds the information needed to create a database using the DBTools library.

Syntax

```
typedef struct a_create_db {
    unsigned short      version;
    const char          *dbname;
    const char          *logname;
    const char          *startline;
    unsigned short      page_size;
    const char          *default_collation;
    const char          *nchar_collation;
    const char          *encoding;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;

    a_bit_field         blank_pad               : 2;
    a_bit_field         respect_case            : 1;
    a_bit_field         encrypt                 : 1;
    a_bit_field         avoid_view_collisions   : 1;
    a_bit_field         jconnect                : 1;
    a_bit_field         checksum                : 1;
    a_bit_field         encrypted_tables        : 1;
    a_bit_field         case_sensitivity_use_default : 1;
    char                verbose;
    char                accent_sensitivity;
    const char          *mirrorname;
    const char          *data_store_type;
    const char          *encryption_key;
    const char          *encryption_algorithm;
    char                *dba_uid;
    char                *dba_pwd;
    unsigned int        db_size;
    int                 db_size_unit;

} a_create_db;
```

Members

Member	Description
version	DBTools version number.
dbname	Database file name.
logname	New transaction log name.

Member	Description

startline	The command line used to start the database server. For example:
	`"d:\sqlany11\bin32\dbeng11.exe"`
	The default start line is used if this member is NULL
	The following is the default START parameter:
	`"dbeng11 -gp page_size -c 10M"`

page_size	The page size of the database.
default_collation	The collation for the database.
nchar_collation	If not NULL, use to generate the NCHAR COLLATION clause with specified string.
errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
blank_pad	Must be one of NO_BLANK_PADDING or BLANK_PADDING. Treat blanks as significant in string comparisons and hold index information to reflect this. See "Blank padding enumeration" on page 871.
respect_case	Make string comparisons case sensitive and hold index information to reflect this.
encrypt	When set, generates the ENCRYPTED ON or, when encrypted_tables is also set, the ENCRYPTED TABLES ON clause.
avoid_view_collisions	Omit the generation of Watcom SQL compatibility views SYS.SYSCOLUMNS and SYS.SYSINDEXES.
jconnect	Include system procedures needed for jConnect.
checksum	Set to 1 for ON or 0 for OFF. Generates one of CHECKSUM ON or CHECKSUM OFF clauses.
encrypted_tables	Set to 1 for encrypted tables. Used with encrypt, generates the ENCRYPTED TABLE ON clause instead of the ENCRYPTED ON clause.
case_sensitivity_use_default	If set, use the default case sensitivity for the locale. This only affects UCA. When set, do not add the CASE RESPECT clause to the CREATE DATABASE statement.
verbose	See "Verbosity enumeration" on page 875.

Member	Description
accent_sensitivity	One of y, n, or f (yes, no, french). Generates one of the ACCENT RESPECT, ACCENT IGNORE or ACCENT FRENCH clauses.
mirrorname	Transaction log mirror name.
data_store_type	Reserved. Use NULL.
encryption_key	The encryption key for the database file. Used with encrypt, it generates the KEY clause.
encryption_algorithm	The encryption algorithm (AES, AES256, AES_FIPS, or AES256_FIPS). Used with encrypt and encryption_key, it generates the ALGORITHM clause.
dba_uid	When not NULL, generates the DBA USER xxx clause.
dba_pwd	When not NULL, generates the DBA PASSWORD xxx clause.
db_size	When not 0, generates the DATABASE SIZE clause.
db_size_unit	Used with db_size, must be one of DBSP_UNIT_NONE, DBSP_UNIT_PAGES, DBSP_UNIT_BYTES, DBSP_UNIT_KILOBYTES, DBSP_UNIT_MEGABYTES, DBSP_UNIT_GIGABYTES, DBSP_UNIT_TERABYTES. When not DBSP_UNIT_NONE, it generates the corresponding keyword (for example, DATABASE SIZE 10 MB is generated when db_size is 10 and db_size_unit is DBSP_UNIT_MEGABYTES). See "Database size unit enumeration" on page 872.

See also

- "DBCreate function" on page 822
- "Using callback functions" on page 815

a_db_info structure

Holds the information needed to return dbinfo information using the DBTools library.

Syntax

```
typedef struct a_db_info {
    unsigned short      version;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    unsigned short      dbbufsize;
    char *              dbnamebuffer;
    unsigned short      logbufsize;
    char *              lognamebuffer;
    unsigned short      mirrorbufsize;
```

```
    char *              mirrornamebuffer;
    unsigned short      collationnamebufsize;
    char *              collationnamebuffer;
    const char *        connectparms;
    a_bit_field         quiet    : 1;
    a_bit_field         page_usage : 1;
    a_sysinfo           sysinfo;
    a_table_info *      totals;
    a_sql_uint32        file_size;
    a_sql_uint32        free_pages;
    a_sql_uint32        bit_map_pages;
    a_sql_uint32        other_pages;
    a_bit_field         checksum : 1;
    a_bit_field         encrypted_tables : 1;
} a_db_info;
```

Members

Member	Description
version	DBTools version number.
errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
statusrtn	Callback routine for handling a status message.
dbbufsize	Set the length of the database file name buffer (for example, _MAX_PATH).
dbnamebuffer	Set the pointer to database file name buffer.
logbufsize	Set the length of the transaction log file name buffer (for example, _MAX_PATH).
lognamebuffer	Set the pointer to the transaction log file name buffer.
mirrorbufsize	Set the length of the mirror file name buffer (for example, _MAX_PATH).
mirrornamebuffer	Set the pointer to the mirror file name buffer.
collationnamebufsize	Set the length of the database collation name and label buffer (the maximum size is 129 including space for the null character).
collationnamebuffer	Set the pointer to the database collation name and label buffer.

Member	Description

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	The database server would be started by the connection string START parameter. For example:
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	A full example connection string including the START parameter:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

quiet	Operate without confirming messages.
page_usage	1 to report page usage statistics, otherwise 0.
sysinfo	a_sysinfo structure (see "a_sysinfo structure" on page 857).
totals	Pointer to a_table_info structure (see "a_table_info structure" on page 858).
file_size	Size of database file.
free_pages	Number of free pages.
bit_map_pages	Number of bitmap pages in the database.
other_pages	Number of pages that are not table pages, index pages, free pages, or bitmap pages.
checksum	Database page checksums enabled if 1, disabled if 0.
encrypted_tables	Encrypted tables are supported if 1, disabled if 0.

See also

- "DBInfo function" on page 823
- "Using callback functions" on page 815

a_db_version_info structure

Holds information regarding which version of SQL Anywhere was used to create the database.

Syntax

```
typedef struct a_db_version_info {
        unsigned short  version;
        const char      *filename;
        a_db_version    created_version;
        MSG_CALLBACK      errorrtn;
        MSG_CALLBACK      msgrtn;
   } a_db_version_info;
```

Members

Member	Description
version	DBTools version number.
filename	Name of the database file to check.
created_version	Set to a value of type a_db_version indicating the server version that create the database file. See “a_db_version enumeration” on page 872.
errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.

See also

- “DBCreatedVersion function” on page 822
- “a_db_version enumeration” on page 872
- “Using callback functions” on page 815

a_dblic_info structure

Holds information containing licensing information. You must use this information only in a manner consistent with your license agreement.

Syntax

```
typedef struct a_dblic_info {
    unsigned short      version;
    char                *exename;
    char                *username;
    char                *compname;
    a_sql_int32         nodecount;
    a_sql_int32         conncount;
    a_license_type      type;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         quiet           : 1;
    a_bit_field         query_only      : 1;
    char                *installkey;
} a_dblic_info;
```


Members

Member	Description
version	DBTools version number.
exename	Name of the server executable or license file.
username	User name for licensing.
compname	Company name for licensing.
nodecount	Number of nodes licensed.
conncount	Must be 1000000L.
type	See *lictype.h* for values.
errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
quiet	Operate without printing messages (1), or print messages (0).
query_only	If 1, just display the license information. If 0, permit changing the information.
installkey	Internal use only. Set to NULL.

a_dbtools_info structure

Holds the information needed to start and finish working with the DBTools library.

Syntax

```
typedef struct a_dbtools_info {
    MSG_CALLBACK        errorrtn;
} a_dbtools_info;
```

Members

Member	Description
errorrtn	Callback routine for handling an error message.

See also

- “DBToolsFini function” on page 826
- “DBToolsInit function” on page 826
- “Using callback functions” on page 815

an_erase_db structure

Holds information needed to erase a database using the DBTools library.

Syntax

```
typedef struct an_erase_db {
    unsigned short      version;
    const char *        dbname;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         quiet : 1;
    a_bit_field         erase : 1;
    const char *        encryption_key;
} an_erase_db;
```

Members

Member	Description
version	DBTools version number.
dbname	Database file name to erase.
confirmrtn	Callback routine for confirming an action.
errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
quiet	Operate without printing messages (1), or print messages (0).
erase	Erase without confirmation (1) or with confirmation (0).
encryption_key	The encryption key for the database file.

See also

- "DBErase function" on page 823
- "Using callback functions" on page 815

a_name structure

Holds a linked list of names. This is used by other structures requiring lists of names.

Syntax

```
typedef struct a_name {
    struct a_name *next;
    char          name[1];
} a_name, * p_name;
```


Members

Member	Description
next	Pointer to the next a_name structure in the list.
name	The name.

See also

- “a_translate_log structure” on page 858
- “a_validate_db structure” on page 869
- “an_unload_db structure” on page 863

a_remote_sql structure

Holds information needed for the dbremote utility using the DBTools library.

Syntax

```
typedef struct a_remote_sql {
    short               version;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_QUEUE_CALLBACK  msgqueuertn;
    char *              connectparms;
    char *              transaction_logs;
    a_bit_field         receive : 1;
    a_bit_field         send : 1;
    a_bit_field         verbose : 1;
    a_bit_field         deleted : 1;
    a_bit_field         apply : 1;
    a_bit_field         batch : 1;
    a_bit_field         more : 1;
    a_bit_field         triggers : 1;
    a_bit_field         debug : 1;
    a_bit_field         rename_log : 1;
    a_bit_field         latest_backup : 1;
    a_bit_field         scan_log : 1;
    a_bit_field         link_debug : 1;
    a_bit_field         full_q_scan : 1;
    a_bit_field         no_user_interaction : 1;
    a_bit_field         _unused1 : 1;
    a_sql_uint32        max_length;
    a_sql_uint32        memory;
    a_sql_uint32        frequency;
    a_sql_uint32        threads;
    a_sql_uint32        operations;
    char *              queueparms;
    char *              locale;
    a_sql_uint32        receive_delay;
    a_sql_uint32        patience_retry;
    MSG_CALLBACK        logrtn;
    a_bit_field         use_hex_offsets : 1;
    a_bit_field         use_relative_offsets : 1;
    a_bit_field         debug_page_offsets : 1;
    a_sql_uint32        debug_dump_size;
```

```
    a_sql_uint32        send_delay;
    a_sql_uint32        resend_urgency;
    char *              include_scan_range;
    SET_WINDOW_TITLE_CALLBACK set_window_title_rtn;
    char *              default_window_title;
    MSG_CALLBACK        progress_msg_rtn;
    SET_PROGRESS_CALLBACK progress_index_rtn;
    char **             argv;
    a_sql_uint32        log_size;
    char *              encryption_key;
    const char *        log_file_name;
    a_bit_field         truncate_remote_output_file:1;
    char *              remote_output_file_name;
    MSG_CALLBACK        warningrtn;
    char *              mirror_logs;
} a_remote_sql;
```

Members

Member	Description
version	DBTools version number.
confirmrtn	Pointer to a function that prints the given message and accepts a yes or no response returning TRUE if yes and FALSE if no.
errorrtn	Pointer to a function that prints the given error message.
msgrtn	Pointer to a function that prints the given informational (non-error) message.
msgqueuertn	Pointer to a function that should sleep for the number of milliseconds passed to it. This function is called with 0 when DBRemoteSQL is busy, but wants to allow the upper layer to process messages. This routine should return MSGQ_SLEEP_THROUGH normally, or MSGQ_SHUTDOWN_REQUESTED to stop SQL Remote processing.

Member	Description

connectparms	Parameters needed to connect to the database. Corresponds to the dbremote -c option. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	The database server would be started by the connection string START parameter. For example:
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	A full example connection string including the START parameter:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

transaction_logs	Pointer to a string naming the directory with offline transaction logs. Corresponds to the transaction_logs_directory argument of dbremote.
receive	If receive is true, messages are received. Corresponds to the dbremote -r option. If receive and send are both false then both are assumed true. It is recommended that you set both to false.
send	If send is true, messages are sent. Corresponds to the dbremote -s option. If receive and send are both false then both are assumed true. It is recommended that you set both to false.
verbose	When true, extra information is printed. Corresponds to the dbremote -v option.
deleted	Should be set to true. If false, messages are not deleted after they are applied. Corresponds to dbremote -p option.
apply	Should be set to true. If false, messages are scanned, but not applied. Corresponds to dbremote -a option.
batch	If true, force exit after applying messages and scanning log. Same as at least one user having 'always' send time. If false, allow run mode to be determined by remote users' send times.
more	Should be set to true.

Member	Description
triggers	Should be set to false in most cases; otherwise, true means DBRemoteSQL replicates trigger actions. Corresponds to the dbremote -t option.
debug	Include debugging output if set true.
rename_log	If set to true, logs are renamed and restarted.
latest_backup	If set to true, only process logs that are backed up. Don't send operations from a live log. Corresponds to the dbremote -u option.
scan_log	Reserved; set to false.
link_debug	If set to true, debugging is turned on for links.
full_q_scan	Reserved; set to false.
no_user_interaction	If set to true, no user interaction is requested.
max_length	Set to the maximum length (in bytes) a message can have. This affects sending and receiving. The recommended value is 50000. Corresponds to the dbremote -l option.
memory	Set to the maximum size (in bytes) of memory buffers to use while building messages to send. The recommended value is at least 2 * 1024 * 1024. Corresponds to the dbremote -m option.
frequency	Set the polling frequency for incoming messages. This value should be set to the max(1, receive_delay/60). See receive_delay below.
threads	Set the number of worker threads that should be used to apply messages. This value must not exceed 50. Corresponds to the dbremote -w option.
operations	This value is used when applying messages. Commits are ignored until DBRemoteSQL has at least this number of operations(inserts, deletes, updates) that are uncommitted. Corresponds to the dbremote -g option.
queueparms	Reserved; set to NULL.
locale	Reserved; set to NULL.
receive_delay	Set this to the time (in seconds) to wait between polls for new incoming messages. The recommended value is 60. Corresponds to the dbremote -rd option.

Member	Description
patience_retry	Set this to the number of polls for incoming messages that DBRemoteSQL should wait before assuming that a message it is expecting is lost. For example, if patience_retry is 3 then DBRemoteSQL tries up to three times to receive the missing message. Afterwards, it sends a resend request. The recommended value is 1. Corresponds to the dbremote -rp option.
logrtn	Pointer to a function that prints the given message to a log file. These messages do not need to be seen by the user.
use_hex_offsets	Set to true if you want log offsets to be shown in hexadecimal notation; otherwise decimal notation will be used.
use_relative_offsets	Set to true if you want log offsets to be displayed as relative to the start of the current log file. Set to false if you want log offsets from the beginning of time to be displayed.
debug_page_offsets	Reserved; set to false.
debug_dump_size	Reserved; set to 0.
send_delay	Set the time (in seconds) between scans of the log file for new operations to send. Set to zero to allow DBRemoteSQL to choose a good value based on user send times. Corresponds to the dbremote -sd option.
resend_urgency	Set the time (in seconds) that DBRemoteSQL waits after seeing that a user needs a rescan before performing a full scan of the log. Set to zero to allow DBRemoteSQL to choose a good value based on user send times and other information it has collected. Corresponds to the dbremote -ru option.
include_scan_range	Reserved; set to NULL.
set_window_title_rtn	Pointer to a function that resets the title of the window (Windows only). The title could be "*database_name* (receiving, scanning, or sending) - *default_window_title*".
default_window_title	A pointer to the default window title string.
progress_msg_rtn	Pointer to a function that displays a progress message.

Member	Description
progress_index_rtn	Pointer to a function that updates the state of the progress bar. This function takes two unsigned integer arguments *index* and *max*. On the first call, the values are the minimum and maximum values (for example, 0, 100). On subsequent calls, the first argument is the current index value (for example, between 0 and 100) and the second argument is always 0.
argv	Pointer to a parsed command line (a vector of pointers to strings). If not NULL, then DBRemoteSQL will call a message routine to display each command line argument except those prefixed with -c, -cq, or -ek.
log_size	DBRemoteSQL renames and restarts the online transaction log when the size of the online transaction log is greater than this value. Corresponds to the dbremote -x option.
encryption_key	Pointer to an encryption key. Corresponds to the dbremote -ek option.
log_file_name	Pointer to the name of the DBRemoteSQL output log to which the message callbacks print their output. If *send* is true, the error log is sent to the consolidated (unless this pointer is NULL).
truncate_remote_output_file	Set to true to cause the remote output file to be truncated rather than appended to. See below. Corresponds to the dbremote -rt option.
remote_output_file_name	Pointer to the name of the DBRemoteSQL remote output file. Corresponds to the dbremote -ro or -rt option.
warningrtn	Pointer to a function that prints the given warning message. If NULL, the *errorrtn* function is called instead.
mirror_logs	Pointer to the name of the directory containing offline mirror transaction logs. Corresponds to the dbremote -ml option.

The dbremote tool sets the following defaults before processing any command-line options:

- version = DB_TOOLS_VERSION_NUMBER
- argv = (argument vector passed to application)
- deleted = TRUE
- apply = TRUE
- more = TRUE
- link_debug = FALSE
- max_length = 50000
- memory = 2 * 1024 * 1024
- frequency = 1
- threads = 0

- receive_delay = 60
- send_delay = 0
- log_size = 0
- patience_retry = 1
- resend_urgency = 0
- log_file_name = (set from command line)
- truncate_remote_output_file = FALSE
- remote_output_file_name = NULL
- no_user_interaction = TRUE (if user interface is not available)
- errorrtn = (address of an appropriate routine)
- msgrtn = (address of an appropriate routine)
- confirmrtn = (address of an appropriate routine)
- msgqueuertn = (address of an appropriate routine)
- logrtn = (address of an appropriate routine)
- warningrtn = (address of an appropriate routine)
- set_window_title_rtn = (address of an appropriate routine)
- progress_msg_rtn = (address of an appropriate routine)
- progress_index_rtn = (address of an appropriate routine)

See also

- "DBRemoteSQL function" on page 825
- "DBTools interface for dbmlsync" [*MobiLink - Client Administration*]

a_sync_db structure

Holds information needed for the dbmlsync utility using the DBTools library.

Syntax

```
typedef struct a_sync_db {
    unsigned short      version;
    char *              connectparms;
    char *              publication;
    const char *        offline_dir;
    char *              extended_options;
    char *              script_full_path;
    const char *        include_scan_range;
    const char *        raw_file;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        logrtn;
    a_sql_uint32        debug_dump_size;
    a_sql_uint32        dl_insert_width;
    a_bit_field         verbose                 : 1;
    a_bit_field         debug                   : 1;
    a_bit_field         debug_dump_hex          : 1;
    a_bit_field         debug_dump_char         : 1;
    a_bit_field         debug_page_offsets      : 1;
    a_bit_field         use_hex_offsets         : 1;
    a_bit_field         use_relative_offsets    : 1;
    a_bit_field         output_to_file          : 1;
    a_bit_field         output_to_mobile_link   : 1;
```

```
a_bit_field         dl_use_put              : 1;
a_bit_field         kill_other_connections  : 1;
a_bit_field         retry_remote_behind     : 1;
a_bit_field         ignore_debug_interrupt  : 1;
SET_WINDOW_TITLE_CALLBACK set_window_title_rtn;
char *              default_window_title;
MSG_QUEUE_CALLBACK  msgqueuertn;
MSG_CALLBACK        progress_msg_rtn;
SET_PROGRESS_CALLBACK progress_index_rtn;
char **             argv;
char **             ce_argv;
a_bit_field         connectparms_allocated  : 1;
a_bit_field         entered_dialog          : 1;
a_bit_field         used_dialog_allocation  : 1;
a_bit_field         ignore_scheduling       : 1;
a_bit_field         ignore_hook_errors      : 1;
a_bit_field         changing_pwd            : 1;
a_bit_field         prompt_again            : 1;
a_bit_field         retry_remote_ahead      : 1;
a_bit_field         rename_log              : 1;
a_bit_field         hide_conn_str           : 1;
a_bit_field         hide_ml_pwd             : 1;
a_sql_uint32        dlg_launch_focus;
char *              mlpassword;
char *              new_mlpassword;
char *              verify_mlpassword;
a_sql_uint32        pub_name_cnt;
char **             pub_name_list;
USAGE_CALLBACK      usage_rtn;
a_sql_uint32        log_size;
a_sql_uint32        hovering_frequency;
a_bit_short         ignore_hovering         : 1;
a_bit_short         verbose_upload          : 1;
a_bit_short         verbose_upload_data     : 1;
a_bit_short         verbose_download        : 1;
a_bit_short         verbose_download_data   : 1;
a_bit_short         autoclose               : 1;
a_bit_short         ping                    : 1;
a_bit_short         _unused                 : 9;
char *              encryption_key;
a_syncpub *         upload_defs;
const char *        log_file_name;
char *              user_name;
a_bit_short         verbose_minimum         : 1;
a_bit_short         verbose_hook            : 1;
a_bit_short         verbose_row_data        : 1;
a_bit_short         verbose_row_cnts        : 1;
a_bit_short         verbose_option_info     : 1;
a_bit_short         strictly_ignore_trigger_ops : 1;
a_bit_short         _unused2                : 10;
a_sql_uint32        est_upld_row_cnt;
STATUS_CALLBACK     status_rtn;
MSG_CALLBACK        warningrtn;
char **             ce_reproc_argv;
a_bit_short         upload_only             : 1;
a_bit_short         download_only           : 1;
a_bit_short         allow_schema_change     : 1;
a_bit_short         dnld_gen_num            : 1;
a_bit_short         _unused3                :12;
const char *        apply_dnld_file;
const char *        create_dnld_file;
char *              sync_params;
const char *        dnld_file_extra;
COMServer *         com_server;
```



```
    a_bit_short         trans_upload            : 1;
    a_bit_short         continue_download       : 1;
    a_bit_short         lite_blob_handling      : 1;
    a_sql_uint32        dnld_read_size;
    a_sql_uint32        dnld_fail_len;
    a_sql_uint32        upld_fail_len;
    a_bit_short         persist_connection      :1;
    a_bit_short         verbose_protocol        :1;
    a_bit_short         no_stream_compress      :1;
    a_bit_short         _unused4                :13;
    const char *        encrypted_stream_opts;
    a_sql_uint32        no_offline_logscan;
    a_bit_short         server_mode :1;
    a_bit_short         allow_outside_connect :1;
    a_bit_short         prompt_for_encrypt_key:1;
    a_bit_short         com_server_mode:1;
    a_bit_short         verbose_server:1;
    a_bit_short         _unused5 :11;
    a_sql_uint32        server_port;
    char *              preload_dlls;
    char *              sync_profile;
    char *              sync_opt;
    a_syncpub *         last_upload_def;
} a_sync_db;
```

Members

Member	Description
version	DBTools version number.

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	The database server would be started by the connection string START parameter. For example:
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	A full example connection string including the START parameter:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

publication	Deprecated; use NULL.
offline_dir	Log directory, as specified on the command line after the options.
extended_options	Extended options, as specified with -e.
script_full_path	Deprecated; use NULL.

Member	Description
include_scan_range	Reserved; use NULL.
raw_file	Reserved; use NULL.
confirmrtn	Reserved; use NULL.
errorrtn	Function to display error messages.
msgrtn	Function to write messages to the user interface and, optionally, to the log file.
logrtn	Function to write messages only to the log file.
debug_dump_size	Reserved; use 0.
dl_insert_width	Reserved; use 0.
verbose	Deprecated; use 0.
debug	Reserved; use 0.
debug_dump_hex	Reserved; use 0.
debug_dump_char	Reserved; use 0.
debug_page_offsets	Reserved; use 0.
use_hex_offsets	Reserved; use 0.
use_relative_offsets	Reserved; use 0.
output_to_file	Reserved; use 0.
output_to_mobile_link	Reserved; use 1.
dl_use_put	Reserved; use 0.
kill_other_connections	TRUE if -d option is specified.
retry_remote_behind	TRUE if -r or -rb is specified.
ignore_debug_interrupt	Reserved; use 0.
set_window_title_rtn	Function to call to change the title of the dbmlsync window (Windows only).
default_window_title	Name of the program to display in the window caption (for example, DBMLSync).

Member	Description
msgqueuertn	Function called by DBMLSync when it wants to sleep. The parameter specifies the sleep period in milliseconds. The function should return the following, as defined in *dllapi.h*. • MSGQ_SLEEP_THROUGH indicates that the routine slept for the requested number of milliseconds. In most cases this is the value you should return. • MSGQ_SHUTDOWN_REQUESTED indicates that you would like the synchronization to terminate as soon as possible. • MSGQ_SYNC_REQUESTED indicates that the routine slept for less than the requested number of milliseconds and that the next synchronization should begin immediately if a synchronization is not currently in progress.
progress_msg_rtn	Function to change the text in the status window, above the progress bar.
progress_index_rtn	Function to update the state of the progress bar.
argv	argv array for this run; the last element of the array must be NULL.
ce_argv	Reserved; use NULL.
connectparms_allocated	Reserved; use 0.
entered_dialog	Reserved; use 0.
used_dialog_allocation	Reserved; use 0.
ignore_scheduling	TRUE if -is was specified.
ignore_hook_errors	TRUE if -eh was specified.
changing_pwd	TRUE if -mn was specified.
prompt_again	Reserved—use 0.
retry_remote_ahead	TRUE if -ra was specified.
rename_log	TRUE if -x was specified, in which case the log file is renamed and restarted.
hide_conn_str	TRUE unless -vc was specified.
hide_ml_pwd	TRUE unless -vp was specified.

Member	Description
dlg_launch_focus	Reserved; use 0.
mlpassword	MobiLink password specified with -mp; NULL otherwise.
new_mlpassword	New MobiLink password specified with -mn; NULL otherwise.
verify_mlpassword	Reserved; use NULL.
pub_name_cnt	Deprecated; use 0.
pub_name_list	Deprecated; use NULL.
usage_rtn	Reserved; use NULL.
log_size	Log size in bytes, as specified with -x; otherwise 0.
hovering_frequency	Hovering frequency in seconds; as set with -pp.
ignore_hovering	True if -p was specified.
verbose_upload	True if -vu was specified.
verbose_upload_data	Reserved; use 0.
verbose_download	Reserved; use 0.
verbose_download_data	Reserved; use 0.
autoclose	TRUE if -k was specified.
ping	TRUE if -pi was specified.
encryption_key	Database key, as specified with -ek.
upload_defs	Linked list of publications to be uploaded together—see a_syncpub.
log_file_name	Database server message log file name specified with -o or -ot.
user_name	MobiLink user name, specified with -u.
verbose_minimum	TRUE if -v was specified.
verbose_hook	TRUE if -vs was specified.
verbose_row_data	TRUE if -vr was specified.
verbose_row_cnts	TRUE if -vn was specified.

Member	Description
verbose_option_info	TRUE if -vo was specified.
strictly_ignore_trigger_ops	Reserved; use 0.
est_upld_row_cnt	Estimated number of rows to upload, specified with -urc.
status_rtn	Reserved; use NULL.
warningrtn	Function to display warning messages.
ce_reproc_argv	Reserved, use NULL.
upload_only	True if -uo was specified.
download_only	TRUE if -ds was specified.
allow_schema_change	TRUE if -sc was specified.
dnld_gen_num	TRUE if -bg was specified.
apply_dnld_file	File specified with -ba; otherwise NULL.
create_dnld_file	File specified with -bc; otherwise NULL.
sync_params	User authentication parameters—specified with -ap.
dnld_file_extra	String specified with -be.
com_server	Reserved; use NULL.
trans_upload	TRUE if -tu was specified.
continue_download	TRUE if -dc is specified.
dnld_read_size	Value specified by -drs option.
dnld_fail_len	Reserved; use 0.
upld_fail_len	Reserved; use 0.
persist_connection	TRUE if -pp is specified on command line.
verbose_protocol	Reserved; use 0.
no_stream_compress	Reserved; use 0.
encrypted_stream_opts	Reserved; use NULL.

Member	Description
no_offline_logscan	TRUE if -do is specified
server_mode	TRUE if -sm specified
allow_outside_connect	Reserved; use 0.
prompt_for_encrypt_key	Reserved; use 0.
com_server_mode	Reserved; use 0.
verbose_server	Reserved; use 0.
server_port	Value from -sp option.
preload_dlls	Reserved; use NULL.
sync_profile	Value specified with -sp option.
sync_opt	Reserved; use NULL.
last_upload_def	Reserved; use NULL.

Some members correspond to features accessible from the dbmlsync command line utility. Unused members should be assigned the value 0, FALSE, or NULL, depending on data type.

See the *dbtools.h* header file for additional comments.

For more information, see "dbmlsync syntax" [*MobiLink - Client Administration*].

See also

- "DBTools interface for dbmlsync" [*MobiLink - Client Administration*]
- "DBSynchronizeLog function" on page 825

a_syncpub structure

Holds information needed for the dbmlsync utility.

Syntax

```
typedef struct a_syncpub {
    struct a_syncpub *  next;
    char *              pub_name;
    char *              ext_opt;
    a_bit_field         alloced_by_dbsync: 1;
} a_syncpub;
```


Members

Member	Description
a_syncpub	Pointer to the next node in the list, NULL for the last node.
pub_name	Publication name(s) specified for this -n option. This is the exact string following -n on the command line.
ext_opt	Extended options specified using the -eu option.
alloced_by_dbsync	Reserved; use FALSE.

See also

- "DBTools interface for dbmlsync" [*MobiLink - Client Administration*]

a_sysinfo structure

Holds information needed for dbinfo and dbunload utilities using the DBTools library.

```
typedef struct a_sysinfo {
    a_bit_field     valid_data          : 1;
    a_bit_field     blank_padding       : 1;
    a_bit_field     case_sensitivity    : 1;
    a_bit_field     encryption          : 1;
    char            default_collation[11];
    unsigned short  page_size;
} a_sysinfo;
```

Members

Member	Description
valid_date	Bit-field indicating whether the following values are set.
blank_padding	1 if blank padding is used in this database, 0 otherwise.
case_sensitivity	1 if the database is case sensitive, 0 otherwise.
encryption	1 if the database is encrypted, 0 otherwise.
default_collation	The collation sequence for the database.
page_size	The page size for the database.

See also

- "a_db_info structure" on page 837

a_table_info structure

Holds information about a table needed as part of the a_db_info structure.

Syntax

```
typedef struct a_table_info {
    struct a_table_info *next;
    a_sql_uint32        table_id;
    a_sql_uint32        table_pages;
    a_sql_uint32        index_pages;
    a_sql_uint32        table_used;
    a_sql_uint32        index_used;
    char *              table_name;
    a_sql_uint32        table_used_pct;
    a_sql_uint32        index_used_pct;
} a_table_info;
```

Members

Member	Description
next	Next table in the list.
table_id	ID number for this table.
table_pages	Number of table pages.
index_pages	Number of index pages.
table_used	Number of bytes used in table pages.
index_used	Number of bytes used in index pages.
table_name	Name of the table.
table_used_pct	Table space utilization as a percentage.
index_used_pct	Index space utilization as a percentage.

See also

- "a_db_info structure" on page 837

a_translate_log structure

Holds information needed for transaction log translation using the DBTools library.

Syntax

```
typedef struct a_translate_log {
    unsigned short      version;
    const char *        connectparms;
```



```
    const char *        logname;
    const char *        sqlname;
    const char *        encryption_key;
    const char *        logs_dir;
    p_name              userlist;
    a_sql_uint32        since_time;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        logrtn;
    MSG_CALLBACK        statusrtn;
    char                userlisttype;
    a_bit_field         quiet                   : 1;
    a_bit_field         remove_rollback         : 1;
    a_bit_field         ansi_sql                : 1;
    a_bit_field         since_checkpoint        : 1;
    a_bit_field         replace                 : 1;
    a_bit_field         include_trigger_trans   : 1;
    a_bit_field         comment_trigger_trans   : 1;
    a_bit_field         debug                   : 1;
    a_bit_field         debug_sql_remote        : 1;
    a_bit_field         debug_dump_hex          : 1;
    a_bit_field         debug_dump_char         : 1;
    a_bit_field         debug_page_offsets      : 1;
    a_bit_field         omit_comments           : 1;
    a_bit_field         use_hex_offsets         : 1;
    a_bit_field         use_relative_offsets    : 1;
    a_bit_field         include_audit           : 1;
    a_bit_field         chronological_order     : 1;
    a_bit_field         force_recovery          : 1;
    a_bit_field         include_subsets         : 1;
    a_bit_field         force_chaining          : 1;
    a_bit_field         generate_reciprocals    : 1;
    a_bit_field         match_mode              : 1;
    a_bit_field         show_undo               : 1;
    a_bit_field         extra_audit             : 1;
    a_sql_uint32        debug_dump_size;
    a_sql_uint32        recovery_ops;
    a_sql_uint32        recovery_bytes;
    const char *        include_source_sets;
    const char *        include_destination_sets;
    const char *        include_scan_range;
    const char *        repserver_users;
    const char *        include_tables;
    const char *        include_publications;
    const char *        queueparms;
    const char *        match_pos;
    a_bit_field         leave_output_on_error   : 1;
} a_translate_log;
```

Members

Member	Description
version	DBTools version number.

Member	Description

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	The database server would be started by the connection string START parameter. For example:
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	A full example connection string including the START parameter:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

logname	Name of the transaction log file. If NULL, there is no log.
sqlname	Name of the SQL output file. If NULL, then name is based on transaction log file name (-n sets this string).
encryption_key	Specify database encryption key (-ek sets string).
logs_dir	Transaction logs directory (-m *dir* sets string); sqlname must be set and connect_parms must be NULL.
userlist	A linked list of user names. Equivalent to -u user1,... or -x user1,... Select or omit transactions for listed users.
since_time	Output from most recent checkpoint prior to time (-j <time> sets this). The number of minutes since January 1, 0001.
confirmrtn	Callback routine for confirming an action.
errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
logrtn	Callback routine to write messages only to the log file.
statusrtn	Callback routine for handling a status message.
userlisttype	Set to DBTRAN_INCLUDE_ALL unless you want to include or exclude a list of users. DBTRAN_INCLUDE_SOME for -u, or DBTRAN_EXCLUDE_SOME for -x.
quiet	Set to TRUE to operate without printing messages (-y).

Member	Description
remove_rollback	Normally set to TRUE; Set to FALSE if you want to include rollback transactions in output (equivalent to -a).
ansi_sql	Set to TRUE if you want to produce ANSI standard SQL transactions (equivalent to -s).
since_checkpoint	Set to TRUE if you want output from most recent checkpoint (equivalent to -f).
replace	Replace existing SQL file without confirmation (equivalent to -y).
include_trigger_trans	Set TRUE to include trigger-generated transactions (equivalent to -g, -sr or -t).
comment_trigger_trans	Set TRUE to include trigger-generated transactions as comments (equivalent to -z).
debug	Reserved; set to FALSE.
debug_sql_remote	Reserved, use FALSE.
debug_dump_hex	Reserved, use FALSE.
debug_dump_char	Reserved, use FALSE.
debug_page_offsets	Reserved, use FALSE.
use_hex_offsets	Reserved, use FALSE.
use_relative_offsets	Reserved, use FALSE.
include_audit	Reserved, use FALSE.
chronological_order	Reserved, use FALSE.
force_recovery	Reserved, use FALSE.
include_subsets	Reserved, use FALSE.
force_chaining	Reserved, use FALSE.
generate_reciprocals	Reserved, use FALSE.
match_mode	Reserved, use FALSE.
show_undo	Reserved, use FALSE.

Member	Description
debug_dump_size	Reserved, use 0.
recovery_ops	Reserved, use 0.
recovery_bytes	Reserved, use 0.
include_source_sets	Reserved, use NULL.
include_destination_sets	Reserved, use NULL.
include_scan_range	Reserved, use NULL.
repserver_users	Reserved, use NULL.
include_tables	Reserved, use NULL.
include_publications	Reserved, use NULL.
queueparms	Reserved, use NULL.
match_pos	Reserved, use NULL.
leave_output_on_error	Set to TRUE if you want to leave the generated .SQL file if corruption detected (equivalent to -k)

The members correspond to features accessible from the dbtran utility.

See the *dbtools.h* header file for additional comments.

See also

- "DBTranslateLog function" on page 827
- "a_name structure" on page 842
- "dbtran_userlist_type enumeration" on page 873
- "Using callback functions" on page 815

a_truncate_log structure

Holds information needed for transaction log truncation using the DBTools library.

Syntax

```
typedef struct a_truncate_log {
    unsigned short      version;
    const char *        connectparms;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         quiet           : 1;
    a_bit_field         server_backup   : 1;
```



```
    char                truncate_interrupted;
} a_truncate_log;
```

Members

Member	Description
version	DBTools version number.

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	The database server would be started by the connection string START parameter. For example:
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	A full example connection string including the START parameter:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
quiet	Operate without printing messages (1), or print messages (0).
server_backup	When set to 1, indicates backup on server using BACKUP DATABASE. Equivalent to dbbackup -s option.
truncate_interrupted	Indicates that the operation was interrupted.

See also

- "DBTruncateLog function" on page 828
- "Using callback functions" on page 815

an_unload_db structure

Holds information needed to unload a database using the DBTools library or extract a remote database for SQL Remote. Those fields used by the dbxtract SQL Remote Extraction utility are indicated.

Syntax

```
typedef struct an_unload_db {
    unsigned short      version;
    const char *        connectparms;
```

```
    const char *        temp_dir;
    const char *        reload_filename;
    char *              reload_connectparms;
    char *              reload_db_filename;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    MSG_CALLBACK        confirmrtn;
    char                unload_type;
    char                verbose;
    char                escape_char;
    char                unload_interrupted;
    a_bit_field         unordered               : 1;
    a_bit_field         no_confirm              : 1;
    a_bit_field         use_internal_unload     : 1;
    a_bit_field         refresh_mat_view        : 1;
    a_bit_field         table_list_provided     : 1;
    a_bit_field         exclude_tables          : 1;
    a_bit_field         preserve_ids            : 1;
    a_bit_field         replace_db              : 1;
    a_bit_short         escape_char_present     : 1;
    a_bit_short         use_internal_reload     : 1;
    a_bit_field         recompute               : 1;
    a_bit_field         make_auxiliary          : 1;
    a_bit_field         encrypted_tables        : 1;
    a_bit_field         remove_encrypted_tables : 1;
    a_bit_field         extract                 : 1;
    a_bit_field         start_subscriptions     : 1;
    a_bit_field         exclude_foreign_keys    : 1;
    a_bit_field         exclude_procedures      : 1;
    a_bit_field         exclude_triggers        : 1;
    a_bit_field         exclude_views           : 1;
    a_bit_field         isolation_set           : 1;
    a_bit_field         include_where_subscribe : 1;
    a_bit_field         exclude_hooks           : 1;
    a_bit_field         startline_name          : 1;
    a_bit_field         debug                   : 1;
    a_bit_field         compress_output         : 1;
    a_bit_field         schema_reload           : 1;
    a_bit_field         genscript               : 1;
    a_bit_field         runscript               : 1;
    const char *        ms_filename;
    int                 ms_reserve;
    int                 ms_size;
    p_name              table_list;
    a_sysinfo           sysinfo;
    const char *        remote_dir;
    const char *        subscriber_username;
    unsigned short      isolation_level;
    const char *        site_name;
    const char *        template_name;
    char *              reload_db_logname;
    const char *        encryption_key;
    const char *        encryption_algorithm;
    unsigned short      reload_page_size;
    const char *        locale;
    const char *        startline;
    const char *        startline_old;
} an_unload_db;
```


Members

Members	Description
version	DBTools version number.

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	The database server would be started by the connection string START parameter. For example:
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	A full example connection string including the START parameter:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

temp_dir	Directory for unloading data files.
reload_filename	The dbunload -r option, something like *reload.sql*.
reload_connectparms	User ID, password, database for reload database.
reload_db_filename	The file name of reload database to create.
errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
statusrtn	Callback routine for handling a status message.
confirmrtn	Callback routine for confirming an action.
unload_type	See "dbunload type enumeration" on page 874.
verbose	See "Verbosity enumeration" on page 875.
escape_char	Used when escape_char_present is TRUE.
unload_interrupted	Set if unload interrupted.
unordered	dbunload -u sets TRUE.
no_confirm	dbunload -y sets TRUE.

Members	Description
use_internal_unload	dbunload -i? sets TRUE. dbunload -x? sets FALSE.
refresh_mat_view	dbunload -g sets TRUE.
table_list_provided	dbunload -e *list* or -i sets TRUE.
exclude_tables	dbunload -e sets TRUE. dbunload -i (undocumented) sets FALSE.
preserve_ids	dbunload sets TRUE/-m sets FALSE.
replace_db	dbunload -ar sets TRUE.
escape_char_present	dbunload -p sets TRUE. Note that escape_char must be set.
use_internal_reload	Usually set TRUE; -ix/-xx sets FALSE; -ii/-xi sets TRUE.
recompute	dbunload -dc sets TRUE. Re-compute all computed columns.
make_auxiliary	dbunload -k sets TRUE. Make auxiliary catalog (for use with diagnostic tracing).
encrypted_tables	dbunload -et sets TRUE. Enable encrypted tables in new database (used with -an or -ar).
remove_encrypted_tables	dbunload -er sets TRUE. Remove encryption from encrypted tables.
extract	TRUE if dbxtract, otherwise FALSE.
start_subscriptions	dbxtract TRUE by default, -b sets FALSE.
exclude_foreign_keys	dbxtract -xf sets TRUE.
exclude_procedures	dbxtract -xp sets TRUE.
exclude_triggers	dbxtract -xt sets TRUE.
exclude_views	dbxtract -xv sets TRUE.
isolation_set	dbxtract -l sets TRUE.
include_where_subscribe	dbxtract -f sets TRUE.
exclude_hooks	dbxtract -hx sets TRUE.
startline_name	(internal use)
debug	(internal use)

Members	Description
compress_output	dbunload -cp sets TRUE.
schema_reload	(internal use)
genscript	(internal use)
runscript	(internal use)
ms_filename	(internal use)
ms_reserve	(internal use)
ms_size	(internal use)
table_list	Selective table list
sysinfo	(internal use)
remote_dir	(like temp_dir) but for internal unloads on server side.
subscriber_username	Argument to dbxtract.
isolation_level	dbxtract -l sets value.
site_name	For dbxtract: specify a site name.
template_name	For dbxtract: specify a template name.
reload_db_logname	Log file name for the reload database.
encryption_key	-ek sets string.
encryption_algorithm	-ea sets one of "AES", "AES256", "AES_FIPS", or "AES256_FIPS".
reload_page_size	dbunload -ap sets value. Set page size of rebuilt database.
locale	(internal use) locale (language and charset).
startline	(internal use)
startline_old	(internal use)

The members correspond to features accessible from the dbunload and dbxtract utilities.

See the *dbtools.h* header file for additional comments.

See also

- "DBUnload function" on page 828

- “a_name structure” on page 842
- “dbunload type enumeration” on page 874
- “Verbosity enumeration” on page 875
- “Using callback functions” on page 815

an_upgrade_db structure

Holds information needed to upgrade a database using the DBTools library.

Syntax

```
typedef struct an_upgrade_db {
    unsigned short      version;
    const char *        connectparms;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         quiet           : 1;
    a_bit_field         jconnect        : 1;
} an_upgrade_db;
```

Members

Member	Description
version	DBTools version number.

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	The database server would be started by the connection string START parameter. For example:
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	A full example connection string including the START parameter:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	For a list of connection parameters, see “Connection parameters” [*SQL Anywhere Server - Database Administration*].

errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
statusrtn	Callback routine for handling a status message.
quiet	Operate without printing messages (1), or print messages (0).

Member	Description
jconnect	Upgrade the database to include jConnect procedures.

See also

- "DBUpgrade function" on page 829
- "Using callback functions" on page 815

a_validate_db structure

Holds information needed for database validation using the DBTools library.

Syntax

```
typedef struct a_validate_db {
    unsigned short      version;
    const char *        connectparms;
    p_name              tables;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         quiet : 1;
    a_bit_field         index : 1;
    a_validate_type     type;
} a_validate_db;
```

Members

Member	Description
version	DBTools version number.

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	The database server would be started by the connection string START parameter. For example:
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	A full example connection string including the START parameter:
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	For a list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

tables	Pointer to a linked list of table names.

Member	Description
errorrtn	Callback routine for handling an error message.
msgrtn	Callback routine for handling an information message.
statusrtn	Callback routine for handling a status message.
quiet	Operate without printing messages (1), or print messages (0).
index	Validate indexes.
type	See “a_validate_type enumeration” on page 874.

See also

- “DBValidate function” on page 829
- “a_name structure” on page 842
- “a_validate_type enumeration” on page 874
- For more information about callback functions, see “Using callback functions” on page 815.

DBTools enumeration types

This section lists the enumeration types that are used by the DBTools library. The enumerations are listed alphabetically.

Blank padding enumeration

Used in the "a_create_db structure" on page 835, to specify the value of blank_pad.

Syntax

```
enum {
    NO_BLANK_PADDING,
    BLANK_PADDING
};
```

Parameters

Value	Description
NO_BLANK_PADDING	Does not use blank padding.
BLANK_PADDING	Uses blank padding.

See also

- "a_create_db structure" on page 835

a_chkpt_log_type enumeration

Used in the "a_backup_db structure" on page 831, to control copying of checkpoint log.

Syntax

```
typedef enum {
    BACKUP_CHKPT_LOG_COPY = 0,
    BACKUP_CHKPT_LOG_NOCOPY,
    BACKUP_CHKPT_LOG_RECOVER,
    BACKUP_CHKPT_LOG_AUTO,
    BACKUP_CHKPT_LOG_DEFAULT
} a_chkpt_log_type;
```

Parameters

Value	Description
BACKUP_CHKPT_LOG_COPY	Use to generate WITH CHECKPOINT LOG COPY clause.
BACKUP_CHKPT_LOG_NOCOPY	Use to generate WITH CHECKPOINT LOG NOCOPY clause.

Value	Description
BACKUP_CHKPT_LOG_RECOVER	Use to generate WITH CHECKPOINT LOG RECOVER clause.
BACKUP_CHKPT_LOG_AUTO	Use to generate WITH CHECKPOINT LOG AUTO clause.
BACKUP_CHKPT_LOG_DEFAULT	Use to omit WITH CHECKPOINT clause.

See also

- "a_backup_db structure" on page 831

a_db_version enumeration

Used in the "a_db_version_info structure" on page 839, to indicate the version of SQL Anywhere that initially created the database.

Syntax

```
enum {
        VERSION_UNKNOWN,
        VERSION_PRE_10,
        VERSION_10,
        VERSION_11
   };
```

Parameters

Value	Description
VERSION_UNKNOWN	Unable to determine the version of SQL Anywhere that created the database.
VERSION_PRE_10	Database was created using a pre-10 version of SQL Anywhere.
VERSION_10	Database was created using SQL Anywhere 10.
VERSION_11	Database was created using SQL Anywhere 11.

See also

- "DBCreatedVersion function" on page 822
- "a_db_version_info structure" on page 839

Database size unit enumeration

Used in the "a_create_db structure" on page 835, to specify the value of db_size_unit.

Syntax

```
enum {
    DBSP_UNIT_NONE,
    DBSP_UNIT_PAGES,
    DBSP_UNIT_BYTES,
    DBSP_UNIT_KILOBYTES,
    DBSP_UNIT_MEGABYTES,
    DBSP_UNIT_GIGABYTES,
    DBSP_UNIT_TERABYTES
};
```

Parameters

Value	Description
DBSP_UNIT_NONE	Units not specified.
DBSP_UNIT_PAGES	Size is specified in pages.
DBSP_UNIT_BYTES	Size is specified in bytes.
DBSP_UNIT_KILOBYTES	Size is specified in kilobytes.
DBSP_UNIT_MEGABYTES	Size is specified in megabytes.
DBSP_UNIT_GIGAYTES	Size is specified in gigabytes.
DBSP_UNIT_TERABYTES	Size is specified in terabytes.

See also

- “a_create_db structure” on page 835

dbtran_userlist_type enumeration

The type of a user list, as used by an “a_translate_log structure” on page 858.

Syntax

```
typedef enum dbtran_userlist_type {
    DBTRAN_INCLUDE_ALL,
    DBTRAN_INCLUDE_SOME,
    DBTRAN_EXCLUDE_SOME
} dbtran_userlist_type;
```

Parameters

Value	Description
DBTRAN_INCLUDE_ALL	Include operations from all users.

Value	Description
DBTRAN_INCLUDE_SOME	Include operations only from the users listed in the supplied user list.
DBTRAN_EXCLUDE_SOME	Exclude operations from the users listed in the supplied user list.

See also

- "a_translate_log structure" on page 858

dbunload type enumeration

The type of unload being performed, as used by the "an_unload_db structure" on page 863.

Syntax

```
enum {
    UNLOAD_ALL,
    UNLOAD_DATA_ONLY,
    UNLOAD_NO_DATA,
    UNLOAD_NO_DATA_FULL_SCRIPT
};
```

Parameters

Value	Description
UNLOAD_ALL	Unload both data and schema.
UNLOAD_DATA_ONLY	Unload data. Do not unload schema. Equivalent to dbunload -d option.
UNLOAD_NO_DATA	No data. Unload schema only. Equivalent to dbunload -n option.
UNLOAD_NO_DATA_FULL_SCRIPT	No data. Include LOAD/INPUT statements in reload script. Equivalent to dbunload -nl option.

See also

- "an_unload_db structure" on page 863

a_validate_type enumeration

The type of validation being performed, as used by the "a_validate_db structure" on page 869.

Syntax

```
typedef enum {
    VALIDATE_NORMAL = 0,
    VALIDATE_DATA,
    VALIDATE_INDEX,
    VALIDATE_EXPRESS,
    VALIDATE_FULL,
    VALIDATE_CHECKSUM,
    VALIDATE_DATABASE,
    VALIDATE_COMPLETE
} a_validate_type;
```

Parameters

Value	Description
VALIDATE_NORMAL	Validate with the default check only.
VALIDATE_DATA	(obsolete)
VALIDATE_INDEX	(obsolete)
VALIDATE_EXPRESS	Validate with express check. Equivalent to dbvalid -fx option.
VALIDATE_FULL	(obsolete)
VALIDATE_CHECKSUM	Validate database checksums. Equivalent to dbvalid -s option.
VALIDATE_DATABASE	Validate database. Equivalent to dbvalid -d option.
VALIDATE_COMPLETE	Perform all possible validation activities.

See also

- “a_validate_db structure” on page 869
- “Validation utility (dbvalid)” [*SQL Anywhere Server - Database Administration*]
- “VALIDATE statement” [*SQL Anywhere Server - SQL Reference*]

Verbosity enumeration

Specifies the volume of output.

Syntax

```
enum {
    VB_QUIET,
    VB_NORMAL,
    VB_VERBOSE
};
```

Parameters

Value	Description
VB_QUIET	No output.
VB_NORMAL	Normal amount of output.
VB_VERBOSE	Verbose output, useful for debugging.

See also

- “a_create_db structure” on page 835
- “an_unload_db structure” on page 863

CHAPTER 23

Exit codes

Contents

Software component exit codes

All database tools are provided as entry points in a DLL. These entry points use the following exit codes. The SQL Anywhere utilities (dbbackup, dbspawn, dbeng11, and so on) also use these exit codes.

Code	Status	Explanation
0	EXIT_OKAY	Success
1	EXIT_FAIL	General failure
2	EXIT_BAD_DATA	Invalid file format
3	EXIT_FILE_ERROR	File not found, unable to open
4	EXIT_OUT_OF_MEMORY	Out of memory
5	EXIT_BREAK	Terminated by the user
6	EXIT_COMMUNICATIONS_FAIL	Failed communications
7	EXIT_MISSING_DATABASE	Missing a required database name
8	EXIT_PROTOCOL_MISMATCH	Client/server protocol mismatch
9	EXIT_UNABLE_TO_CONNECT	Unable to connect to the database server
10	EXIT_ENGINE_NOT_RUNNING	Database server not running
11	EXIT_SERVER_NOT_FOUND	Database server not found
12	EXIT_BAD_ENCRYPT_KEY	Missing or bad encryption key
13	EXIT_DB_VER_NEWER	Server must be upgraded to run database
14	EXIT_FILE_INVALID_DB	File is not a database
15	EXIT_LOG_FILE_ERROR	Log file was missing or other error
16	EXIT_FILE_IN_USE	File in use
17	EXIT_FATAL_ERROR	Fatal error or assertion occurred
255	EXIT_USAGE	Invalid parameters on the command line

These exit codes are contained in the *install-dir\sdk\include\sqldef.h* file.

Part V. Deploying SQL Anywhere

This part introduces you to deployment strategies for SQL Anywhere.

CHAPTER 24

Deploying databases and applications

Contents

Introduction to deployment

When you have completed a database application, you must deploy the application to your end users. Depending on the way in which your application uses SQL Anywhere (as an embedded database, in a client/server fashion, and so on) you may have to deploy components of the SQL Anywhere software along with your application. You may also have to deploy configuration information, such as data source names, that enable your application to communicate with SQL Anywhere.

Check your license agreement
Redistribution of files is subject to your license agreement with Sybase. No statements in this document override anything in your license agreement. Check your license agreement before considering deployment.

The following deployment steps are examined in this chapter:

- Determining required files based on the choice of application platform and architecture.
- Configuring client applications.

Much of the chapter deals with individual files and where they need to be placed. However, the recommended way of deploying SQL Anywhere components is to use the **Deployment Wizard** or to use a silent install. For information, see "Using the Deployment Wizard" on page 887 and "Using a silent install for deployment" on page 889.

Types of deployment

The files you need to deploy depend on the type of deployment you choose. Here are some possible deployment models:

- **Client deployment** You may deploy only the client portions of SQL Anywhere to your end users, so that they can connect to a centrally located network database server.
- **Network server deployment** You may deploy network servers to offices, and then deploy clients to each of the users within those offices.
- **Embedded database deployment** You may deploy an application that runs with the personal database server. In this case, both client and personal server need to be installed on the end-user's computer.
- **SQL Remote deployment** Deploying a SQL Remote application is an extension of the embedded database deployment model.
- **MobiLink deployment** For information about deploying MobiLink servers, see "Deploying MobiLink applications" [*MobiLink - Server Administration*].
- **Administration tools deployment** You may deploy Interactive SQL, Sybase Central and other management tools.

Ways to distribute files

There are two ways to deploy SQL Anywhere:

- **Use the SQL Anywhere installer** You can make the installer available to your end users. By selecting the proper option, each end user is guaranteed to receive the files they need.

  This is the simplest solution for many deployment cases. In this case, you must still provide your end users with a method for connecting to the database server (such as an ODBC data source).

  For more information, see "Using the Deployment Wizard" on page 887 or "Using a silent install for deployment" on page 889.

- **Develop your own installation** There may be reasons for you to develop your own installation program that includes SQL Anywhere files. This is a more complicated option, and most of this chapter addresses the needs of those who are developing their own installation.

  If SQL Anywhere has already been installed for the server type and operating system required by the client application architecture, the required files can be found in the appropriately-named subdirectory, located in the SQL Anywhere installation directory. For example, the *bin32* subdirectory of your installation directory contains the files required to run the server for 32-bit Windows operating systems.

Whichever option you choose, you must not violate the terms of your license agreement.

Understanding installation directories and file names

For a deployed application to work properly, the database server and client applications must each be able to locate the files they need. The deployed files should be located relative to each other in the same fashion as your SQL Anywhere installation.

In practice, this means that on Windows, most files belong in a single directory. For example, on Windows both client and database server required files are installed in a single directory, which is the *bin32* subdirectory of the SQL Anywhere installation directory.

For a full description of the places where the software looks for files, see "How SQL Anywhere locates files" [*SQL Anywhere Server - Database Administration*].

Linux/Unix/Mac OS X deployment issues

Unix deployments are different from Windows deployments in some ways:

- **Directory structure** For Linux/Unix/Mac OS X installations, the directory structure is as follows:

Directory	Contents
/opt/sqlanywhere11/bin	Executable files, License files
/opt/sqlanywhere11/lib	Shared objects and libraries
/opt/sqlanywhere11/res	String files

  On AIX, the default root directory is */usr/lpp/sqlanywhere11* instead of */opt/sqlanywhere11*.

  On Mac OS X, the default root directory is */Applications/SQLAnywhere11/System* instead of */opt/sqlanywhere11*.

- **File suffixes** In the tables in this chapter, the shared objects are listed with a suffix of *.so* or *.so.1*. The version number, 1, could be higher as updates are released. For simplicity, the version number is often not listed.

  For AIX, the suffix does not contain a version number so it is simply *.so*.

- **Symbolic links** Each shared object is installed as a symbolic link (symlink) to a file of the same name with the additional suffix *.1* (one). For example, *libdblib11.so* is a symbolic link to the file *libdblib11.so.1* in the same directory.

  The version suffix *.1* could be higher as updates are released and the symbolic link must be redirected, accordingly.

- **Threaded and non-threaded applications** Most shared objects are provided in two forms, one of which has the additional characters *_r* before the file suffix. For example, in addition to *libdblib11.so.1*, there is a file named *libdblib11_r.so.1*. In this case, threaded applications must be linked to the shared object whose name has the *_r* suffix, while non-threaded applications must be linked to the shared object

whose name does not have the *_r* suffix. Occasionally, there is a third form of shared object with *_n* before the file suffix. This is a version of the shared object that is used with non-threaded applications.

- **Character set conversion** If you want to use database server character set conversion, you need to include the following files:
  - *libdbicu11.so.1*
  - *libdbicu11_r.so.1*
  - *libdbicudt11.so.1*
  - *sqlany.cvf*
- **Environment variables** On Linux/Unix, environment variables must be set for the system to be able to locate SQL Anywhere applications and libraries. It is recommended that you use the appropriate file for your shell, either *sa_config.sh* or *sa_config.csh* (located in the directory */opt/sqlanywhere11/bin*) as a template for setting the required environment variables. Some of the environment variables set by these files include PATH, LD_LIBRARY_PATH, and SQLANY11.

For a description of how SQL Anywhere looks for files, see "How SQL Anywhere locates files" [*SQL Anywhere Server - Database Administration*].

File naming conventions

SQL Anywhere uses consistent file naming conventions to help identify and group system components.

These conventions include:

- **Version number** The SQL Anywhere version number is indicated in the file name of the main server components (executable files, dynamic link libraries, shared objects, license files, and so on).

  For example, the file *dbeng11.exe* is a version 11 executable for Windows.
- **Language** The language used in a language resource library is indicated by a two-letter code within its file name. The two characters before the version number indicate the language used in the library. For example, *dblgen11.dll* is the message resource library for the English language. These two-letter codes are specified by ISO standard 639.

  For more information about language labels, see "Language Selection utility (dblang)" [*SQL Anywhere Server - Database Administration*].

  For a list of the languages available in SQL Anywhere, see "Localized versions of SQL Anywhere" [*SQL Anywhere Server - Database Administration*].

Identifying other file types

The following table identifies the platform and function of SQL Anywhere files according to their file extension. SQL Anywhere follows standard file extension conventions where possible.

File extension	Platform	File type
.bat, .cmd	Windows	Batch command files
.chm, .chw	Windows	Help system file

File extension	Platform	File type
.dll	Windows	Dynamic Link Library
.exe	Windows	Executable file
.ini	All	Initialization file
.lic	All	License file
.lib	Varies by development tool	Static runtime libraries for the creation of embedded SQL executables
.res	Linux/Unix, Mac OS X	Language resource file for non-Windows environments
.so	Linux/Unix	Shared object or shared library file. The equivalent of a Windows DLL
.bundle, .dylib	Mac OS X	Shared object file. The equivalent of a Windows DLL

Database file names

SQL Anywhere databases are composed of two elements:

- **Database file** This is used to store information in an organized format. By default, this file uses a *.db* file extension. There may also be additional dbspace files. These files could have any file extension including none.
- **Transaction log file** This is used to record all changes made to data stored in the database file. By default, this file uses a *.log* file extension, and is generated by SQL Anywhere if no such file exists and a log file is specified to be used. A mirrored transaction log has the default extension of *.mlg*.

These files are updated, maintained and managed by the SQL Anywhere relational database management system.

Using the Deployment Wizard

The SQL Anywhere **Deployment Wizard** is the preferred tool for creating 32-bit deployments of SQL Anywhere for Windows. The **Deployment Wizard** can create installer files that include some or all of the following components:

- Client interfaces such as ODBC
- SQL Anywhere server, including remote data access, database tools, and encryption
- UltraLite relational database
- MobiLink server, client, Monitor, and encryption
- QAnywhere messaging
- Administration tools such as Interactive SQL and Sybase Central

The **Deployment Wizard** does not include support for creating deployments of the 64-bit software components.

You can use the **Deployment Wizard** to create a Microsoft Windows Installer Package file or a Microsoft Windows Installer Merge Module file:

- **Microsoft Windows Installer Package file** A storage file containing the instructions and data required to install an application. An Installer Package file has the extension *.msi*.
- **Microsoft Windows Installer Merge Module file** A simplified type of Microsoft Installer Package file that includes all files, resources, registry entries, and setup logic to install a shared component. A merge module has the extension *.msm*.

  A merge module cannot be installed alone because it lacks some vital database tables that are present in an installer package file. Merge modules also contain additional tables that are unique to themselves. To install the information delivered by a merge module with an application, the module must first be merged into the application's Installer Package (*.msi*) file. A merge module consists of the following parts:

  - A merge module database containing the installation properties and setup logic being delivered by the merge module.
  - A merge module Summary Information Stream describing the module.
  - A *MergeModule.CAB* cabinet file stored as a stream inside the merge module. This cabinet contains all the files required by the components delivered by the merge module. Every file delivered by the merge module must be stored inside of a cabinet file that is embedded as a stream in the merge module's structured storage. In a standard merge module, the name of this cabinet is always: *MergeModule.CAB*.

Note
Redistribution of files is subject to your license agreement. You must acknowledge that you are properly licensed to redistribute SQL Anywhere files. Check your license agreement before proceeding.

To create a deployment file

1. Start the **Deployment Wizard**:

- From the **Start** menu, choose **Programs** » **SQL Anywhere 11** » **Deploy SQL Anywhere For Windows**.

  or

- From the *Deployment* subdirectory of your SQL Anywhere installation, run *DeploymentWizard.exe*.

2. Follow the instructions in the wizard.

To install a deployment file

- Use the Microsoft Windows Installer to install the deployment file. Here is a sample command:

  ```
  msiexec /package sqlany11.msi
  ```

  A silent install can be performed using a command like the following:

  ```
  msiexec /qn /package sqlany11.msi SQLANYDIR=c:\sa11
  ```

  - **/package <package-name>** This parameter tells the Microsoft Windows Installer to install the specified package (in this case, *sqlany11.msi*).
  - **/qn** This parameter tells the Microsoft Windows Installer to operate in the background with no user interaction.
  - **SQLANYDIR** The value of this parameter is the path to where the software is to be installed.

To uninstall a deployment

- It is also possible to perform a silent uninstall. The following is an example of a command line that would do this.

  ```
  msiexec /uninstall sqlany11.msi
  ```

  Alternately, a product code can be specified.

  ```
  msiexec.exe /qn /uninstall {19972A31-72EF-126F-31C7-5CF249B8593F}
  ```

  - **/qn** This parameter tells the Microsoft Windows Installer to operate in the background with no user interaction.
  - **/uninstall <package-name> | <product-code>** This parameter tells the Microsoft Windows Installer to uninstall the product associated with the specified MSI file or product code.

For more tips on how to do silent installs, see "Using a silent install for deployment" on page 889.

Using a silent install for deployment

Silent installs run without user input and with no indication to the user that an install is occurring. On Windows operating systems, you can call the SQL Anywhere installer from your own install program in such a way that the SQL Anywhere install is silent.

The options for the SQL Anywhere install program *setup.exe* are:

- **/L:language_id** The language identifier is a locale number that represents the language for the install. For example, locale ID 1033 identifies U.S. English, locale ID 1031 identifies German, locale ID 1036 identifies French, locale ID 1041 identifies Japanese, and locale ID 2052 identifies Simplified Chinese.
- **/S** This option hides the initialization dialogue. Use this option in conjunction with **/V**.
- **/V** Specify parameters to MSIEXEC, the Microsoft Windows Installer tool.

The following command line example assumes that the install image directory is in the *software\sqlanywhere* directory on the CD in drive *d:*.

```
d:\software\sqlanywhere\setup.exe /l:1033 /s "/v:/qn
    REGKEY=PEPEV-E96QE-A4000-00000-00000 INSTALLDIR=c:\sa11 DIR_SAMPLES=c:
\sa11\Samples"
```

- **REGKEY** The value of this parameter must be a valid software installation key.
- **INSTALLDIR** The value of this parameter is the path to where the software is installed.
- **DIR_SAMPLES** The value of this parameter is the path to where the sample programs are installed.
- **USERNAME** The value of this parameter is the user name to record for this installation (for example, USERNAME=\"John Smith\").
- **COMPANYNAME** The value of this parameter is the company name to record for this installation (for example, COMPANYNAME=\"Smith Holdings\").

An example of a command line that specifies all of the above options is:

```
d:\software\sqlanywhere\setup.exe /l:1033 /s "/v:/qn
    REGKEY=PEPEV-E96QE-A4000-00000-00000
    INSTALLDIR=c:\sa11
    DIR_SAMPLES=c:\sa11\Samples"
    USERNAME=\"John Smith\"
    COMPANYNAME=\"Smith Holdings\"
```

Although the above text is shown over several lines for reasons of length, it would be specified as a single line of text. Note the use of the backslash character to escape the interior quotation marks.

In addition to a silent install, it is also possible to perform a silent uninstall. The following is an example of a command line that would do this.

```
msiexec.exe /qn /uninstall {ECE263B0-6C8B-404C-B4AC-8FAB1C87AB4A}
```

In the above example, you call the Microsoft Windows Installer tool directly.

- **/qn** This parameter tells the Microsoft Windows Installer to operate in the background with no user interaction.

- **/uninstall <product-code>** This parameter tells the Microsoft Windows Installer to uninstall the product associated with the specified product code.

The silent install described above does not address how you would select a subset of the components to install. This topic is better addressed by the **Deployment Wizard**. For information on component selection, see "Using the Deployment Wizard" on page 887.

Deploying client applications

To deploy a client application that runs against a network database server, you must provide each end user with the following items:

- **Client application** The application software itself is independent of the database software, and so is not described here.
- **Database interface files** The client application requires the files for the database interface it uses (.NET, ADO, OLE DB, ODBC, JDBC, embedded SQL, or Open Client).
- **Connection information** Each client application needs database connection information.

The interface files and connection information required varies with the interface your application is using. Each interface is described separately in the following sections.

The simplest way to deploy clients is to use the **Deployment Wizard**. For more information, see "Using the Deployment Wizard" on page 887.

Deploying .NET clients

The simplest way to deploy .NET assemblies is to use the **Deployment Wizard**. For more information, see "Using the Deployment Wizard" on page 887.

If you want to create your own installation, this section describes the files to deploy to the end users.

Each .NET client computer must have the following:

- **A working .NET installation** Microsoft .NET assemblies and instructions for their redistribution are available from Microsoft Corporation. They are not described in detail here.
- **The SQL Anywhere .NET Data Provider** The following table shows the files needed for a working SQL Anywhere .NET data provider. These files should be placed in a single directory.

  The SQL Anywhere installation places the Windows assembly for the .NET Framework in the *Assembly\v2* subdirectory of your SQL Anywhere installation directory. The other files are placed in the operating-system subdirectory of your SQL Anywhere installation directory (for example, *bin32* or *bin64*).

  The SQL Anywhere installation places the Windows Mobile assemblies for the .NET Compact Framework in *ce\Assembly\v2*. The other file is placed in the Windows Mobile subdirectory of your SQL Anywhere installation directory (for example, *ce\arm.50*).

Description	Windows	Windows Mobile
.NET driver file	*iAnywhere.Data.SQLAnywhere.dll*	*iAnywhere.Data.SQLAnywhere.dll*
.NET Global Assembly Cache	N/A	*iAnywhere.Data.SQLAnywhere.gac*

Description	Windows	Windows Mobile
Language-resource library	*dblg[en]11.dll*	*dblg[en]11.dll*
Connect window	*dbcon11.dll*	N/A

For more information about deploying the SQL Anywhere .NET provider, see "Deploying the SQL Anywhere .NET Data Provider" on page 137.

Deploying OLE DB and ADO clients

The simplest way to deploy OLE DB client libraries is to use the **Deployment Wizard**. For more information, see "Using the Deployment Wizard" on page 887.

If you want to create your own installation, this section describes the files to deploy to the end users.

Each OLE DB client computer must have the following:

- **A working OLE DB installation** OLE DB files and instructions for their redistribution are available from Microsoft Corporation. They are not described in detail here.
- **The SQL Anywhere OLE DB provider** The following table shows the files needed for a working SQL Anywhere OLE DB provider. These files should be placed in a single directory. The SQL Anywhere installation places them all in the operating-system subdirectory of your SQL Anywhere installation directory (for example, *bin32* or *bin64*). For Windows, there are two provider DLLs. The second DLL (*dboledba11*) is an assist DLL used to provide schema support. There is no second DLL for Windows Mobile.

Description	Windows	Windows Mobile
OLE DB driver file	*dboledb11.dll*	*dboledb11.dll*
OLE DB driver file	*dboledba11.dll*	N/A
Language-resource library	*dblg[en]11.dll*	*dblg[en]11.dll*
Connect window	*dbcon11.dll*	N/A
Elevated operations agent	*dbelevate11.exe*	N/A

OLE DB providers require many registry entries. You can make these by self-registering the DLLs using the regsvr32 utility on Windows or the regsvrce utility on Windows Mobile.

Note that for Windows Vista or later versions of Windows, you must include the SQL Anywhere elevated operations agent which supports the privilege elevation required when DLLs are registered or unregistered. This file is only required as part of the OLE DB provider install or uninstall procedure.

For Windows clients, it is recommended that you use Microsoft MDAC 2.7 or later.

When deploying OLE DB applications to Windows Mobile devices, you must also include Microsoft's ADOCE version 3.1 (or later). At minimum, the following files are required.

```
adoce31.dll
adocedb31.dll
adoceoledb31.dll
msdadc.dll
msdaer.dll
msdaerXX.dll (where XX is the 2 letter language code, EN for English)
```

The Microsoft Data Access Components (MDAC) files *msdadc.dll* and *msdaer.dll* must be registered on the device.

For more information, see "Creating a Windows Mobile database" [*SQL Anywhere Server - Database Administration*].

Customizing the OLE DB provider

When installing the OLE DB provider, the Windows Registry must modified. Typically, this is done using the self-registration capability built into the OLE DB provider. For example, you would use the Windows regsvr32 tool to do this. A standard set of registry entries are created by the provider.

In a typical connection string, one of the components is the Provider attribute. To indicate that the SQL Anywhere OLE DB provider is to be used, you specify the name of the provider. Here is a Visual Basic example:

```
connectString = "Provider=SAOLEDB;DSN=SQL Anywhere 11 Demo"
```

With ADO and/or OLE DB, there are many other ways to reference the provider by name. Here is a C++ example in which you specify not only the provider name but also the version to use.

```
hr = db.Open(_T("SAOLEDB.11"), &dbinit);
```

The provider name is looked up in the registry. If you were to examine the registry on your computer system, you would find an entry in HKEY_CLASSES_ROOT for SAOLEDB.

```
[HKEY_CLASSES_ROOT\SAOLEDB]
@="SQL Anywhere OLE DB Provider"
```

It has two subkeys that contain a class identifier (ClsId) and current version (CurVer) for the provider. Here is an example.

```
[HKEY_CLASSES_ROOT\SAOLEDB\Clsid]
@="{41dfe9f3-db91-11d2-8c43-006008d26a6f}"

[HKEY_CLASSES_ROOT\SAOLEDB\CurVer]
@="SAOLEDB.11"
```

There are several more similar entries. They are used to identify a specific instance of an OLE DB provider. If you look up the ClsId in the registry under HKEY_CLASSES_ROOT\CLSID and examine the subkeys, you see that one of the entries identifies the location of the provider DLL.

```
[HKEY_CLASSES_ROOT\CLSID\
{41dfe9f3-db91-11d2-8c43-006008d26a6f}\
```

```
InprocServer32]

@="c:\\sa11\\bin64\\dboledb11.dll"
"ThreadingModel"="Both"
```

The problem here is that the structure is very monolithic. If you were to uninstall the SQL Anywhere software from your system, the OLE DB provider registry entries would be removed from your registry and then the provider DLL would be removed from your hard drive. Any applications that depend on the provider would no longer work.

Similarly, if applications from different vendors all use the same OLE DB provider, then each installation of the same provider would overwrite the common registry settings. The version of the provider that you intended your application to work with would be supplanted by another newer (or older!) version of the provider.

Clearly, the instability that could arise from this situation is undesirable. To address this problem, the SQL Anywhere OLE DB provider can be customized.

In the following exercise, you generate a unique set of GUIDs, choose a unique provider name and unique DLL names. These three things will help you create a unique OLE DB provider which you can deploy with your application.

Here are the steps involved in creating a custom version of the OLE DB provider.

To customize the OLE DB provider

1. Make a copy of the sample registration file shown below. It is listed after these steps because it is quite lengthy. The file name should have a *.reg* suffix. The names of the registry values are case sensitive.
2. Create 4 sequential UUIDs (GUIDs).

   Use the uuidgen tool from Microsoft's Visual Studio.

   ```
   uuidgen -n4 -s -x >oledbguids.txt
   ```
3. The 4 UUIDs or GUIDs are assigned in sequence as follows.
   a. The Provider class ID (GUID1 below).
   b. The Enum class ID (GUID2 below).
   c. The ErrorLookup class ID (GUID3 below).
   d. The Provider Assist class ID (GUID4 below). This last GUID is not used in Windows Mobile deployments.

   It is important that they be sequential (that is what -x in the uuidgen command line does for you). Each GUID should look something like the following.

Name	GUID
GUID1	41dfe9f3-db92-11d2-8c43-006008d26a6f
GUID2	41dfe9f4-db92-11d2-8c43-006008d26a6f
GUID3	41dfe9f5-db92-11d2-8c43-006008d26a6f

Name	GUID
GUID4	41dfe9f6-db92-11d2-8c43-006008d26a6f

Note that it is the first part of the GUID (for example, 41dfe9f3) that is incrementing.

4. Use the search/replace capability of an editor to change all the GUID1, GUID2, GUID3, and GUID4 in the text to the corresponding GUID (for example, GUID1 would be replaced by 41dfe9f3-db92-11d2-8c43-006008d26a6f if that was the GUID generated for you by uuidgen).
5. Decide on your Provider name. This is the name that you will use in your application in connection strings, and so on (for example, Provider=SQLAny). Do not use any of the following names. These names are used by SQL Anywhere.

Version 10 or later	Version 9 or earlier
SAOLEDB	ASAProv
SAErrorLookup	ASAErrorLookup
SAEnum	ASAEnum
SAOLEDBA	ASAProvA

6. Use the search/replace capability of an editor to change all the occurrences of the string SQLAny to the provider name that you have chosen. This includes all those places where SQLAny may be a substring of a longer string (for example, SQLAnyEnum).

   Suppose you chose Acme for your provider name. The names that will appear in the HKEY_CLASSES_ROOT registry hive are shown in the following table along with the SQL Anywhere names (for comparison).

SQL Anywhere provider	Your custom provider
SAOLEDB	Acme
SAErrorLookup	AcmeErrorLookup
SAEnum	AcmeEnum
SAOLEDBA	AcmeA

7. Make copies of the SQL Anywhere provider DLLs (*dboledb11.dll* and *dboledba11.dll*) under different names. Note that there is no *dboledba11.dll* for Windows Mobile.

   ```
   copy dboledb11.dll myoledb11.dll
   copy dboledba11.dll myoledba11.dll
   ```

   A special registry key will be created by the script that is based on the DLL name that you choose. It is important that the name be different from the standard DLL names (such as *dboledb11.dll* or *dboledba11.dll*). If you name the provider DLL *myoledb11* then the provider will look up a registry entry in HKEY_CLASSES_ROOT with that same name. The same is true of the provider schema assist DLL.

If you name the DLL *myoledba11* then the provider will look up a registry entry in HKEY_CLASSES_ROOT with that same name. It is important that the name you choose is unique and is unlikely to be chosen by anyone else. Here are some examples.

DLL name(s) chosen	Corresponding HKEY_CLASSES_ROOT\name
myoledb11.dll	HKEY_CLASSES_ROOT\myoledb11
myoledba11.dll	HKEY_CLASSES_ROOT\myoledba11
acmeOledb.dll	HKEY_CLASSES_ROOT\acmeOledb
acmeOledba.dll	HKEY_CLASSES_ROOT\acmeOledba
SAcustom.dll	HKEY_CLASSES_ROOT\SAcustom
SAcustomA.dll	HKEY_CLASSES_ROOT\SAcustomA

8. Use the search/replace capability of an editor to change all the occurrences of *myoledb11* and *myoledba11* in the registry script to the two DLL names you have chosen.
9. Use the search/replace capability of an editor to change all the occurrences of *d:\\mypath\\bin32* in the registry script to the installed location for the DLLs. Be sure to use a pair of slashes to represent a single slash. This step will have to be customized at the time of your application install.
10. Save the registry script to disk and run it.
11. Give your new provider a try. Do not forget to change your ADO / OLE DB application to use the new provider name.

Here is the listing of the registry script that is to be modified.

```
REGEDIT4
; Special registry entries for a private OLE DB provider.

[HKEY_CLASSES_ROOT\myoledb11]
@="Custom SQL Anywhere OLE DB Provider 11.0"

[HKEY_CLASSES_ROOT\myoledb11\Clsid]
@="{GUID1}"

; Data1 of the following GUID must be 3 greater than the
; previous, for example, 41dfe9f3 + 3 => 41dfe9ee.

[HKEY_CLASSES_ROOT\myoledba11]
@="Custom SQL Anywhere OLE DB Provider 11.0"

[HKEY_CLASSES_ROOT\myoledba11\Clsid]
@="{GUID4}"

; Current version (or version independent prog ID)
; entries (what you get when you have "SQLAny"
; instead of "SQLAny.11")

[HKEY_CLASSES_ROOT\SQLAny]
@="SQL Anywhere OLE DB Provider"
```



```
[HKEY_CLASSES_ROOT\SQLAny\Clsid]
@="{GUID1}"

[HKEY_CLASSES_ROOT\SQLAny\CurVer]
@="SQLAny.11"

[HKEY_CLASSES_ROOT\SQLAnyEnum]
@="SQL Anywhere OLE DB Provider Enumerator"

[HKEY_CLASSES_ROOT\SQLAnyEnum\Clsid]
@="{GUID2}"

[HKEY_CLASSES_ROOT\SQLAnyEnum\CurVer]
@="SQLAnyEnum.11"

[HKEY_CLASSES_ROOT\SQLAnyErrorLookup]
@="SQL Anywhere OLE DB Provider Extended Error Support"

[HKEY_CLASSES_ROOT\SQLAnyErrorLookup\Clsid]
@="{GUID3}"

[HKEY_CLASSES_ROOT\SQLAnyErrorLookup\CurVer]
@="SQLAnyErrorLookup.11"

[HKEY_CLASSES_ROOT\SQLAnyA]
@="SQL Anywhere OLE DB Provider Assist"

[HKEY_CLASSES_ROOT\SQLAnyA\Clsid]
@="{GUID4}"

[HKEY_CLASSES_ROOT\SQLAnyA\CurVer]
@="SQLAnyA.11"

; Standard entries (Provider=SQLAny.11)

[HKEY_CLASSES_ROOT\SQLAny.11]
@="Sybase SQL Anywhere OLE DB Provider 11.0"

[HKEY_CLASSES_ROOT\SQLAny.11\Clsid]
@="{GUID1}"

[HKEY_CLASSES_ROOT\SQLAnyEnum.11]
@="Sybase SQL Anywhere OLE DB Provider Enumerator 11.0"

[HKEY_CLASSES_ROOT\SQLAnyEnum.11\Clsid]
@="{GUID2}"

[HKEY_CLASSES_ROOT\SQLAnyErrorLookup.11]
@="Sybase SQL Anywhere OLE DB Provider Extended Error Support 11.0"

[HKEY_CLASSES_ROOT\SQLAnyErrorLookup.11\Clsid]
@="{GUID3}"

[HKEY_CLASSES_ROOT\SQLAnyA.11]
@="Sybase SQL Anywhere OLE DB Provider Assist 11.0"

[HKEY_CLASSES_ROOT\SQLAnyA.11\Clsid]
@="{GUID4}"

; SQLAny (Provider=SQLAny.11)

[HKEY_CLASSES_ROOT\CLSID\{GUID1}]
@="SQLAny.11"
"OLEDB_SERVICES"=dword:ffffffff
```

```
[HKEY_CLASSES_ROOT\CLSID\{GUID1}\ExtendedErrors]
@="Extended Error Service"

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\ExtendedErrors\{GUID3}]
@="Sybase SQL Anywhere OLE DB Provider Error Lookup"

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\InprocServer32]
@="d:\\mypath\\bin32\\myoledb11.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\OLE DB Provider]
@="Sybase SQL Anywhere OLE DB Provider 11.0"

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\ProgID]
@="SQLAny.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\VersionIndependentProgID]
@="SQLAny"

; SQLAnyErrorLookup

[HKEY_CLASSES_ROOT\CLSID\{GUID3}]
@="Sybase SQL Anywhere OLE DB Provider Error Lookup 11.0"
@="SQLAnyErrorLookup.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID3}\InprocServer32]
@="d:\\mypath\\bin32\\myoledb11.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\CLSID\{GUID3}\ProgID]
@="SQLAnyErrorLookup.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID3}\VersionIndependentProgID]
@="SQLAnyErrorLookup"

; SQLAnyEnum

[HKEY_CLASSES_ROOT\CLSID\{GUID2}]
@="SQLAnyEnum.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID2}\InprocServer32]
@="d:\\mypath\\bin32\\myoledb11.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\CLSID\{GUID2}\OLE DB Enumerator]
@="Sybase SQL Anywhere OLE DB Provider Enumerator"

[HKEY_CLASSES_ROOT\CLSID\{GUID2}\ProgId]
@="SQLAnyEnum.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID2}\VersionIndependentProgID]
@="SQLAnyEnum"

; SQLAnyA

[HKEY_CLASSES_ROOT\CLSID\{GUID4}]
@="SQLAnyA.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID4}\InprocServer32]
@="d:\\mypath\\bin32\\myoledba11.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\CLSID\{GUID4}\ProgID]
```



```
@="SQLAnyA.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID4}\VersionIndependentProgID]
@="SQLAnyA"
```

Deploying ODBC clients

The simplest way to deploy ODBC clients is to use the **Deployment Wizard**. For more information, see "Using the Deployment Wizard" on page 887.

Each ODBC client computer must have the following:

- **ODBC Driver Manager** Microsoft provides an ODBC Driver Manager for Windows operating systems. SQL Anywhere includes an ODBC Driver Manager for Unix. There is no ODBC Driver Manager for Windows Mobile. ODBC applications can run without a driver manager but, on platforms for which an ODBC driver manager is available, this is not recommended.
- **Connection information** The client application must have access to the information needed to connect to the server. This information is typically included in an ODBC data source.
- **The SQL Anywhere ODBC driver** The files that must be included in a deployment of an ODBC client application are described next in ODBC driver required files.

ODBC driver required files

The following table shows the files needed for a working SQL Anywhere ODBC driver. These files should be placed in a single directory. The SQL Anywhere installation places them all in the operating-system subdirectory of your SQL Anywhere installation directory (for example, *bin32* or *bin64*).

The multithreaded version of the ODBC driver for Unix platforms is indicated by "MT".

Platform	Required files
Windows	*dbodbc11.dll* *dbcon11.dll* *dbicu11.dll* *dbicudt11.dll* *dblg[en]11.dll* *dbelevate11.exe*
Windows Mobile	*dbodbc11.dll* *dbicu11.dll* (optional) *dbicudt11.dat* (optional) *dblg[en]11.dll*

Platform	Required files
Linux/Solaris	*libdbodbc11.so.1* *libdbodbc11_n.so.1* *libdbodm11.so.1* *libdbtasks11.so.1* *libdbicu11.so.1* *libdbicudt11.so.1* *dblg[en]11.res*
Linux/Solaris MT	*libdbodbc11.so.1* *libdbodbc11_r.so.1* *libdbodm11.so.1* *libdbtasks11_r.so.1* *libdbicu11_r.so.1* *libdbicudt11.so.1* *dblg[en]11.res*
HP-UX	*libdbodbc11.so.1* *libdbodbc11_n.so.1* *libdbodm11.so.1* *libdbtasks11.so.1* *libdbicu11.so.1* *libdbicudt11.so.1* *dblg[en]11.res*
HP-UX MT	*libdbodbc11.so.1* *libdbodbc11_r.so.1* *libdbodm11.so.1* *libdbtasks11_r.so.1* *libdbicu11_r.so.1* *libdbicudt11.so.1* *dblg[en]11.res*

Platform	Required files
AIX	*libdbodbc11.so* *libdbodbc11_n.so* *libdbodm11.so* *libdbtasks11.so* *libdbicu11.so* *libdbicudt11.so* *dblg[en]11.res*
AIX MT	*libdbodbc11.so* *libdbodbc11_r.so* *libdbodm11.so* *libdbtasks11_r.so* *libdbicu11_r.so* *libdbicudt11.so* *dblg[en]11.res*
Mac OS X	*dbodbc11.bundle* *libdbodbc11.dylib* *libdbodbc11_n.dylib* *libdbodm11.dylib* *libdbtasks11.dylib* *libdbicu11.dylib* *libdbicudt11.dylib* *dblg[en]11.res*

Platform	Required files
Mac OS X MT	*dbodbc11_r.bundle* *libdbodbc11.dylib* *libdbodbc11_r.dylib* *libdbodm11.dylib* *libdbtasks11_r.dylib* *libdbicu11_r.dylib* *libdbicudt11.dylib* *dblg[en]11.res*

Notes

- There are multithreaded (MT) versions of the ODBC driver for Unix platforms. The file names contain the "_r" suffix. Deploy these files if your application requires them. Also, for Unix platforms, you should create a link to these files. The link name should match the file name with the ".1" version suffix removed.
- For Windows, a driver manager is included with the operating system. For Unix, SQL Anywhere provides a driver manager. The file name begins with *libdbodm11*.
- Note that for Windows Vista or later versions of Windows, you must include the SQL Anywhere elevated operations agent (*dbelevate11.exe*) which supports the privilege elevation required in order to register or unregister the ODBC driver. This file is only required as part of the ODBC driver install or uninstall procedure.
- A language resource library file should also be included. The table lists the English "en" version. Deploy the language resource libraries that correspond to the languages you want to support.
- For Windows, the Connect window support code (*dbcon11.dll*) is needed if your end users will create their own data sources, if they need to enter user IDs and passwords when connecting to the database, or if they need to display the Connect window for any other purpose.

Configuring the ODBC driver

In addition to copying the ODBC driver files onto disk, your installation program must also make a set of registry entries to install the ODBC driver properly.

Windows

The SQL Anywhere installer makes changes to the Windows Registry to identify and configure the ODBC driver. If you are building an installation program for your end users, you should make the same registry settings.

The simplest way to do this is to use the self-registering capability of the ODBC driver. You use the regsvr32 utility on Windows or the regsvrce utility on Windows Mobile. Note that for 64-bit versions of Windows,

you can register both the 64-bit and 32-bit versions of the ODBC driver. By using the self-registering feature of the ODBC driver, you are ensured that the proper registry entries are created.

You can use the regedit utility to inspect the registry entries created by the ODBC driver.

The SQL Anywhere ODBC driver is identified to the system by a set of registry values in the following registry key:

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
      ODBC\
         ODBCINST.INI\
            SQL Anywhere 11
```

Sample values for 32-bit Windows are shown below:

Value name	Value type	Value data
Driver	String	*install-dir\bin32\dbodbc11.dll*
Setup	String	*install-dir\bin32\dbodbc11.dll*

There is also a registry value in the following key:

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
      ODBC\
         ODBCINST.INI\
            ODBC Drivers
```

The value is as follows:

Value name	Value type	Value data
SQL Anywhere 11	String	Installed

64-bit Windows

For 64-bit Windows, the 32-bit ODBC driver registry entries ("SQL Anywhere 11" and "ODBC Drivers") will be located under the following key:

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
      Wow6432Node\
         ODBC\
            ODBCINST.INI
```

To view these entries, you must be using a 64-bit version of regedit. If you cannot locate Wow6432Node on 64-bit Windows, then you are using the 32-bit version of regedit.

Third party ODBC drivers

If you are using a third-party ODBC driver on an operating system other than Windows, consult the documentation for that driver on how to configure the ODBC driver.

Deploying connection information

ODBC client connection information is generally deployed as an ODBC data source. You can deploy an ODBC data source in one of the following ways:

- **Programmatically** Add a data source description to your end-user's registry or ODBC initialization files.
- **Manually** Provide your end users with instructions, so that they can create an appropriate data source on their own computer.

  You create a data source manually using the ODBC Administrator, from the User DSN tab or the System DSN tab. The SQL Anywhere ODBC driver displays the configuration window for entering settings. Data source settings include the location of the database file, the name of the database server, as well as any start up parameters and other options.

This section provides you with the information you need to know for either approach.

Types of data source

There are three kinds of data sources: User data sources, System data sources, and File data sources.

User data source definitions are stored in the part of the registry containing settings for the specific user currently logged on to the system. System data sources, however, are available to all users and to Windows services, which run regardless of whether a user is logged onto the system or not. Given a correctly configured System data source named MyApp, any user can use that ODBC connection by providing DSN=MyApp in the ODBC connection string.

File data sources are not held in the registry, but are held in a special directory. A connection string must provide a FileDSN connection parameter to use a File data source.

Data source registry entries

Each user data source is identified to the system by registry entries. The simplest way to ensure the correct creation of registry entries for data source definitions is to use the SQL Anywhere dbdsn utility to create them.

Otherwise, you must create a set of registry values in a particular registry key.

For User data sources, the key is as follows:

```
HKEY_CURRENT_USER\
   SOFTWARE\
      ODBC\
         ODBC.INI\
            user-data-source-name
```

For System data sources, the key is as follows:

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
      ODBC\
         ODBC.INI\
            system-data-source-name
```


The key contains a set of registry values, each of which corresponds to a connection parameter. For example, the SQL Anywhere 11 Demo key corresponding to the SQL Anywhere 11 Demo system Data Source Name (DSN) contains the following settings for 32-bit Windows:

Value name	Value type	Value data
Autostop	String	YES
DatabaseFile	String	*samples-dir\demo.db*
Description	String	SQL Anywhere 11 Sample Database
Driver	String	*install-dir\bin32\dbodbc11.dll*
ServerName	String	demo11
Integrated	String	NO
PWD	String	sql
Start	String	*install-dir\bin32\dbeng11.exe*
UID	String	DBA

Note
It is recommended that you include the ServerName parameter in connection strings for deployed applications. This ensures that the application connects to the correct server in the event that a computer is running multiple SQL Anywhere database servers and can help prevent timing-dependent connection failures.

In these entries, *install-dir* is the SQL Anywhere installation directory. For 64-bit Windows, *bin32* would be replaced by *bin64*.

In addition, you must add the data source name to the list of data sources in the registry. For user data sources, you use the following key:

```
HKEY_CURRENT_USER\
   SOFTWARE\
      ODBC\
         ODBC.INI\
            ODBC Data Sources
```

For system data sources, use the following key:

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
      ODBC\
         ODBC.INI\
            ODBC Data Sources
```

The value associates each data source with an ODBC driver. The value name is the data source name, and the value data is the ODBC driver name. For example, the system data source installed by SQL Anywhere is named SQL Anywhere 11 Demo, and has the following value:

Value name	Value type	Value data
SQL Anywhere 11 Demo	String	SQL Anywhere 11

Caution: ODBC settings are easily viewed
User data source configurations can contain sensitive database settings such as a user's ID and password. These settings are stored in the registry in plain text, and can be viewed using the Windows Registry editors *regedit.exe* or *regedt32.exe*, which are provided by Microsoft with the operating system. You can choose to encrypt passwords, or require users to enter them when connecting.

Required and optional connection parameters

You can identify the data source name in an ODBC configuration string in this manner,

```
DSN=UserDataSourceName
```

When a DSN parameter is provided in the connection string, the Current User data source definitions in the Windows Registry are searched, followed by System data sources. File data sources are searched only when FileDSN is provided in the ODBC connection string.

The following table illustrates the implications to the user and developer when a data source exists and is included in the application's connection string as a DSN or FileDSN parameter.

When the data source ...	The connection string must also identify ...	The user must supply ...
Contains the ODBC driver name and location; the name of the database file/server; startup parameters; and the user ID and password.	No additional information	No additional information.
Contains the ODBC driver name and location; the name of the database file/server; startup parameters.	No additional information	User ID and password if not provided in the DSN.
Contains only the name and location of the ODBC driver.	The name of the database file (DBF=) and/or the database server (ENG=). Optionally, it may contain other connection parameters such as Userid (UID=) and PASSWORD (PWD=).	User ID and password if not provided in the DSN or ODBC connection string.

When the data source ...	The connection string must also identify ...	The user must supply ...
Does not exist	The name of the ODBC driver to be used (Driver=) and the database name (DBN=), the database file (DBF=), and/or the database server (ENG=). Optionally, it may contain other connection parameters such as Userid (UID=) and PASSWORD (PWD=).	User ID and password if not provided in the ODBC connection string.

For more information about ODBC connections and configurations, see the following:

- "Connecting to a database" [*SQL Anywhere Server - Database Administration*].
- The Open Database Connectivity (ODBC) SDK, available from Microsoft.

Deploying embedded SQL clients

The simplest way to deploy embedded SQL clients is to use the **Deployment Wizard**. For more information, see "Using the Deployment Wizard" on page 887.

Deploying embedded SQL clients involves the following:

- **Installed files** Each client computer must have the files required for a SQL Anywhere embedded SQL client application.
- **Connection information** The client application must have access to the information needed to connect to the server. This information may be included in an ODBC data source.

Installing files for embedded SQL clients

The following table shows which files are needed for embedded SQL clients.

Description	Windows	Unix
Interface library	*dblib11.dll*	*libdblib11.so*, *libdbtasks11.so*
Language resource library	*dblg[en]11.dll*	*dblg[en]11.res*
Connect window	*dbcon11.dll*	N/A

Notes

- If the client application uses encryption then the appropriate encryption support (*dbecc11.dll*, *dbfips11.dll*, or *dbrsa11.dll*) should also be included.

- If the client application uses an ODBC data source to hold the connection parameters, your end user must have a working ODBC installation. Instructions for deploying ODBC are included in the Microsoft ODBC SDK.

  For more information about deploying ODBC information, see "Deploying ODBC clients" on page 899.
- The Connect window support (*dbcon11.dll*) is needed if your end users will be creating their own data sources, if they will need to enter user IDs and passwords when connecting to the database, or if they need to display the Connect window for any other purpose.
- For multi-threaded applications on Unix, use *libdblib11_r.so* and *libdbtasks11_r.so*.

Connection information

You can deploy embedded SQL connection information in one of the following ways:

- **Manual** Provide your end users with instructions for creating an appropriate data source on their computer.
- **File** Distribute a file that contains connection information in a format that your application can read.
- **ODBC data source** You can use an ODBC data source to hold connection information.

Deploying JDBC clients

You must install a Java Runtime Environment to use JDBC.

You may need to define the JAVA_HOME environment variable, if it has not already been defined, so that the database server can locate the Java VM. The JAVA_HOME or JAVAHOME environment variables are commonly created when installing a Java VM. This environment variable should point to the root directory of your Java VM.

The database server searches for the Java VM in the following order:

1. Check the java_location database option.
2. Check the JAVA_HOME environment variable.
3. Check the JAVAHOME environment variable.
4. Check the path.
5. If these searches fail, then the database server fails to load the Java VM.

If the JAVAHOME environment variable is defined and the client computer has JAVA_HOME already defined, the server uses JAVA_HOME.

In addition to a Java Runtime Environment, each JDBC client requires the iAnywhere JDBC driver or jConnect.

iAnywhere JDBC driver

To deploy the iAnywhere JDBC driver, you must deploy the following files:

- *jodbc.jar* This must be in the application's classpath. This file is located in the SQL Anywhere installation *java* folder.
- *dbjodbc11.dll* This must be locatable in the system path. On Unix environments, the file is a shared library called *libdbjodbc11.so*. On MAC OS X, the file is a shared library called *libdbjodbc11.jnilib*.
- The ODBC driver files. For more information, see "ODBC driver required files" on page 899.

jConnect JDBC driver

To deploy the jConnect JDBC driver, you must deploy the following files:

- The jConnect driver files. For a version of the jConnect software and the jConnect documentation, see jConnect for JDBC.
- When you use a TDS client (either Open Client or jConnect based), you have the option of sending the connection password in clear text or in encrypted form. The latter is done by performing a TDS encrypted password handshake. The handshake involves using private/public key encryption. The support for generating the RSA private/public key pair and for decrypting the encrypted password is included in a special library. The library file must locatable in the system path. For Windows, this file is called *dbrsakp11.dll*. There are both 64-bit and 32-bit versions of the DLL. On Unix environments, the file is a shared library called *libdbrsakp11.so*. On MAC OS X, the file is a shared library called *libdbrsakp11.jnilib*. The file is not necessary if you do not use this feature.

JDBC database connection URL

Your Java application needs a URL to connect to the database. This URL specifies the driver, the computer to use, and the port on which the database server is listening.

For more information about URLs, see "Supplying a URL to the driver" on page 485.

Deploying Open Client applications

To deploy Open Client applications, each client computer needs the Sybase Open Client product. You must purchase the Open Client software separately from Sybase. It contains its own installation instructions.

When you use a TDS client (either Open Client or jConnect based), you have the option of sending the connection password in clear text or in encrypted form. The latter is done by performing a TDS encrypted password handshake. The handshake involves using private/public key encryption. The support for generating the RSA private/public key pair and for decrypting the encrypted password is included in a special library. The library file must locatable in the system path. For Windows, this file is called *dbrsakp11.dll*. There are both 64-bit and 32-bit versions of the DLL. On Unix environments, the file is a shared library called *libdbrsakp11.so*. On MAC OS X, the file is a shared library called *libdbrsakp11.jnilib*. The file is not necessary if you do not use this feature.

Connection information for Open Client clients is held in the interfaces file. For information about the interfaces file, see the Open Client documentation and "Configuring Open Servers" [*SQL Anywhere Server - Database Administration*].

Deploying administration tools

Subject to your license agreement, you can deploy a set of administration tools including Interactive SQL, Sybase Central, and the SQL Anywhere Console utility.

The simplest way to deploy the administration tools is to use the **Deployment Wizard**. For more information, see "Using the Deployment Wizard" on page 887.

For information about system requirements for administration tools, see SQL Anywhere Supported Platforms and Engineering Status.

Deployment of the administration tools can be simplified by the use of initialization files. Each of the launcher executables for the administration tools (Sybase Central, Interactive SQL, the Console utility, and MobiLink Monitor) can have a corresponding *.ini* file. This eliminates the need for registry entries and a fixed directory structure for the location of the JAR files. Each launcher executable requires its own *.ini* file. These are located in the same directory and with the same file name as the executable file.

- **dbconsole.ini** This is the name of the Console utility initialization file.
- **dbisql.ini** This is the name of the Interactive SQL initialization file.
- **mlmon.ini** This is the name of the MobiLink Monitor initialization file.
- **scjview.ini** This is the name of the Sybase Central initialization file.

The initialization file will contain the details on how to load the database administration tool. For example, the initialization file may contain some or all of the following lines:

- **JRE_DIRECTORY=<path>** This is the location of the required JRE. The **JRE_DIRECTORY** specification is required.
- **VM_ARGUMENTS=<any required VM arguments>** VM arguments are separated by semicolons (;). Any path values that contain blanks should be enclosed in quotation marks. VM arguments can be discovered by using the `-batch` option of the administration tool (for example, `scjview -batch`) and examining the file that is created. The **VM_ARGUMENTS** specification is optional.
- **JAR_PATHS=<path1;path2;...>** A delimited list of directories which contain the JAR files for the program. They are separated by semicolons (;). The **JAR_PATHS** specification is optional.
- **ADDITIONAL_CLASSPATH=<path1;path2;...>** Classpath values are separated by semicolons (;). The **ADDITIONAL_CLASSPATH** specification is optional.
- **LIBRARY_PATHS=<path1;path2;...>** These are paths to the DLLs/shared objects. They are separated by semicolons (;). The **LIBRARY_PATHS** specification is optional.
- **APPLICATION_ARGUMENTS=<arg1;arg2;...>** These are any application arguments. They are separated by semicolons (;). Application arguments can be discovered by using the `-batch` option of the administration tool (for example, `scjview -batch`) and examining the file that is created. The **APPLICATION_ARGUMENTS** specification is optional.

Here are the contents of a sample initialization file for Sybase Central.

```
JRE_DIRECTORY=c:\Sun\JRE160_x86
VM_ARGUMENTS=-Xmx200m
JAR_PATHS=c:\scj\jars;c:\scj\jhelp
```



```
ADDITIONAL_CLASSPATH=
LIBRARY_PATHS=c:\scj\bin
APPLICATION_ARGUMENTS=-screpository=C:\Documents and Settings\All Users
\Application Data\Sybase Central 6.0.0;-installdir=c:\scj
```

This scenario assumes that a copy of the 32-bit Sun JRE is located in *c:\Sun\JRE160_x86*. As well, the Sybase Central executable and shared libraries (DLLs) like *jsyblib600* are stored in *c:\scj\bin*. The SQL Anywhere JAR files are stored in *c:\scj\jars*. The Sun JavaHelp 2.0 JAR files are stored in *c:\scj\jhelp*.

Note
When you are deploying applications, the personal database server (dbeng11) is required for creating databases using the dbinit utility. It is also required if you are creating databases from Sybase Central on the local computer when no other database servers are running.

Deploying administration tools on Windows without InstallShield

This section explains how to install Interactive SQL (dbisql), Sybase Central (including the SQL Anywhere, MobiLink, QAnywhere and UltraLite plug-ins), and the SQL Anywhere Console utility (dbconsole) on a Windows computer without using InstallShield. It is intended for those who want to create an installer for these administration tools.

This information applies to all Windows platforms except Windows Mobile. The instructions given here are specific to version 11.0.0 and cannot be applied to earlier or later versions of the software.

Check your license agreement
Redistribution of files is subject to your license agreement. No statements in this document override anything in your license agreement. Check your license agreement before considering deployment.

Before you begin

Before reading this section, you should have an understanding of the Windows Registry, including the REGEDIT application. The names of the registry values are case sensitive.

Modifying your registry is dangerous
Modify your registry at your own risk. It is recommended that you back up your system before modifying the registry.

The following steps are required to deploy the administration tools:

1. Decide what you want to deploy.
2. Copy the required files.
3. Register the administration tools with Windows.
4. Update the system path.

5. Register the plug-ins with Sybase Central.
6. Register the SQL Anywhere ODBC driver with Windows.
7. Register the online help files with Windows.

Each of these steps is explained in detail in the following sections.

Step 1: Deciding what software to deploy

You can install any combination of the following software bundles:

- Interactive SQL
- Sybase Central with the SQL Anywhere plug-in
- Sybase Central with the MobiLink plug-in
- Sybase Central with the QAnywhere plug-in
- Sybase Central with the UltraLite plug-in
- SQL Anywhere Console utility (dbconsole)

The following components are also required when installing any of the above software bundles:

- The SQL Anywhere ODBC Driver
- The Java Runtime Environment (JRE) version 1.6.0

Note
To check your JRE version on Mac OS X, to go to the **Apple** menu, and then choose **System Preferences** » **Software Updates**. Click **Installed Updates** for a list of updates that have been applied. If Java 1.6.0 is not in the list, go to developer.apple.com/java/download/.

The instructions in the following sections are structured so that you can install any (or all) of these six bundles without conflicts.

Step 2: Copying the required files

The administration tools require a specific directory structure. You are free to put the directory tree in any directory, on any drive. Throughout the following discussion, *c:\sa11* is used as the example installation folder. The software must be installed into a directory tree structure having the following layout:

Directory	Description
sa11	The root folder. While the following steps assume you are installing into *c:\sa11*, you are free to put the directory anywhere (for example, *C:\Program Files\SQLAny11*).

Directory	Description
sa11\java	Holds Java program JAR files.
sa11\bin32	Holds the native 32-bit Windows components used by the program, including the programs that launch the applications.
sa11\Sun\JavaHelp-2_0	The JavaHelp runtime library.
sa11\Sun\jre160_x86	The 32-bit Java Runtime Environment.

x64
The Java-based administration tools are 32-bit applications. There are no 64-bit versions. The 32-bit administration tools can be deployed on the Windows x64-based platform.

Itanium 64
There are no Java-based administration tools available for deployment on the Itanium (ia64) platform. However, there is a native version of Interactive SQL that is not as feature-rich as the Java version. See "Deploying dbisqlc" on page 932.

The following table lists the files required for each of the software bundles. Make a list of the files you need, and then copy them into the directory structure outlined above. In general, you should take the files from an already-installed copy of SQL Anywhere.

File	Interactive SQL	Sybase Central with the SQL Anywhere plug-in	Sybase Central with the MobiLink plug-in	Sybase Central with the QAnywhere plug-in	Sybase Central with the UltraLite plug-in	SQL Anywhere Console
documentation\[en]\htmlhelp\sqlanywhere_[en]11.chm	X	X	X	X	X	X
\documentation\[en]\htmlhelp\dbadmin_[en]11.map	X	X	X	X	X	X
c:\windows\system32\keyHH.exe[1]	X	X	X	X	X	X
java\jodbc.jar	X	X	X	X	X	X
java\JComponents1100.jar	X	X	X	X	X	X
java\jlogon.jar	X	X	X	X	X	X

File	Interactive SQL	Sybase Central with the SQL Anywhere plug-in	Sybase Central with the MobiLink plug-in	Sybase Central with the QAnywhere plug-in	Sybase Central with the UltraLite plug-in	SQL Anywhere Console
java\SCEditor600.jar	X	X	X	X	X	X
java\jsyblib600.jar	X	X	X	X	X	X
bin32\jsyblib600.dll	X	X	X	X	X	X
Sun\JavaHelp-2_0\jh.jar	X	X	X	X	X	X
Sun\jre160_x86\...	X	X	X	X	X	X
bin32\dblib11.dll	X	X	X	X	X	X
bin32\dbjodbc11.dll	X	X	X	X	X	X
bin32\dbodbc11.dll	X	X	X	X	X	X
bin32\dbcon11.dll	X	X	X	X	X	X
bin32\dblg[en]11.dll	X	X	X	X	X	X
bin32\dbtool11.dll	X	X	X			
bin32\dbisql.com	X					
bin32\dbisql.exe	X					
java\isql.jar	X	X	X		X	
java\saip11.jar	X	X	X		X	
bin32\scjview.exe		X	X	X	X	
bin32\scvw[en]600.jar		X	X	X	X	
java\sybasecentral600.jar		X	X	X	X	
java\salib.jar		X	X	X	X	
java\saplugin.jar		X				
java\debugger.jar		X				

File	Interactive SQL	Sybase Central with the SQL Anywhere plug-in	Sybase Central with the MobiLink plug-in	Sybase Central with the QAnywhere plug-in	Sybase Central with the UltraLite plug-in	SQL Anywhere Console
bin32\dbput11.dll		X	X			
java\apache_files.txt			X	X		
java\apache_license_1.1.txt			X	X		
java\apache_license_2.0.txt			X	X		
java\log4j.jar			X	X		
java\mlplugin.jar			X			
java\mldesign.jar			X			
java\stax-api-1.0.jar			X			
java\wstx-asl-2.0.5.jar			X			
java\velocity.jar			X			
java\velocity-dep.jar			X			
java\qaplugin.jar				X		
java\qaconnector.jar				X		
java\mlstream.jar				X		
bin32\qaagent.exe				X		
bin32\dbicu11.dll				X		
bin32\dbicudt11.dll				X		
bin32\dbghelp.dll				X		
bin32\dbinit.exe				X		
java\ulplugin.jar					X	
bin32\dbconsole.exe						X

File	Inter-active SQL	Sybase Central with the SQL Any-where plug-in	Sybase Central with the Mobi-Link plug-in	Sybase Central with the QAny-where plug-in	Sybase Central with the UltraLite plug-in	SQL Any-where Con-sole
java\DBConsole.jar						X

[1] The exact name of the Windows system directory differs depending on which operating system you are using.

The table above shows a number of files with the designation **[en]**. These files are available in a number of different languages with English (**en**) being just one of them. For more information, see the following section International message and context-sensitive help files.

Some file paths above end with "...". This indicates that the entire tree, including subdirectories, should be copied.

The administration tools require JRE 1.6.0. You should not substitute a later patch version of the JRE unless you have a specific need to do so. Copy the 32-bit version of the JRE files from the *install-dir\Sun \jre160_x86* directory. Copy the entire *jre160_x86* tree, including subdirectories.

For reference, the *sqlanywhere.jpr* file contains a list of the required component files for the SQL Anywhere plug-in of Sybase Central.

The *mobiLink.jpr* file contains a list of the required component files for the MobiLink plug-in of Sybase Central.

The *qanywhere.jpr* file contains a list of the required component files for the QAnywhere plug-in of Sybase Central. When deploying the QAnywhere plug-in, dbinit is required. For information about deploying database tools, see “Deploying database utilities” on page 940.

The *ultralite.jpr* file contains a list of the required component files for the UltraLite plug-in of Sybase Central.

International message and context-sensitive help files

All displayed text and context-sensitive help for the administration tools is translated from English into French, German, Japanese, and Simplified Chinese. The resources for each language are held in separate files. The English files contain **en** in the file names. French files have similar names, but use **fr** instead of **en**. German file names contain **de**, Japanese file names contain **ja**, and Chinese file names contain **zh**.

If you want to install support for different languages, you have to add the message files for those other languages. The translated files are as follows:

dblgen11.dll	English
dblgde11.dll	German
dblges11.dll	Spanish

dblgfr11.dll	French
dblgit11.dll	Italian
dblgja11.dll	Japanese
dblgko11.dll	Korean
dblglt11.dll	Lithuanian
dblgpl11.dll	Polish
dblgpt11.dll	Portuguese
dblgru11.dll	Russian
dblgtw11.dll	Traditional Chinese
dblguk11.dll	Ukrainian
dblgzh11.dll	Simplified Chinese

You must also add the context-sensitive help files for those other languages. The available translated files are as follows:

scvwen600.jar	English
scvwde600.jar	German
scvwfr600.jar	French
scvwja600.jar	Japanese
scvwzh600.jar	Simplified Chinese

These files are included with localized versions of SQL Anywhere.

Step 3: Registering the administration tools with Windows

You must set the following registry values for the administration tools. The names of the registry values are case sensitive.

- In **HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\Sybase Central\6.0.0**
  - **Language** The two-letter code for the language used by Sybase Central. This must be one of the following: **EN**, **DE**, **FR**, **JA**, or **ZH** for English, French, German, Japanese, and Simplified Chinese, respectively.
- In **HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\SQL Anywhere\11.0**

- **Location** The fully-qualified path to the root of the installation folder (*C:\Program Files\SQL Anywhere 11* by default) containing the Sybase Central files.
- **Language** The two-letter code for the language used by SQL Anywhere. This must be one of the following: **EN**, **DE**, **FR**, **JA**, or **ZH** for English, French, German, Japanese, and Simplified Chinese, respectively.

On 64-bit Windows, these registry entries are in the 32-bit registry (**SOFTWARE\Wow6432Node\Sybase**).

Paths should *not* end in a backslash.

Your installer can encapsulate all this information by creating a *.reg* file and then executing it. Using our example installation folder of *c:\sa11*, the following is a sample *.reg* file:

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\Sybase Central\6.0.0]
"Language"="EN"

[HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\SQL Anywhere\11.0]
"Location"="c:\\sa11"
"Language"="EN"
```

Backslashes in file paths must be escaped by another backslash in a *.reg* file.

Step 4: Updating the system path

To run the administration tools, the directories with *.exe* and *.dll* files must be included in the path. You must add the *c:\sa11\bin32* directory to the system path.

On Windows, the system path is stored in the following registry key:

```
HKEY_LOCAL_MACHINE\
  SYSTEM\
    CurrentControlSet\
      Control\
        Session Manager\
          Environment\
            Path
```

When you deploy Interactive SQL or Sybase Central, add the following directory to the existing path:

```
c:\sa11\bin32
```

Step 5: Registering the Sybase Central plug-ins

This step involves the configuration of Sybase Central. If you are not installing Sybase Central, you can skip it.

Sybase Central requires a configuration file that lists the installed plug-ins. Your installer must create this file. Note that it contains full paths to a number of JAR files that may change depending on where the software is installed.

The file is called *.scRepository600*. On Windows XP/200x, it resides in the *%allusersprofile%\application data\Sybase Central 6.0.0* folder. On Windows Vista, it resides in the *%ProgramData%\Sybase Central 6.0.0* folder. It is a plain text file that contains some basic information about the plug-ins that Sybase Central should load.

On Windows Vista, all users should have read access to the directory that contains the *.scRepository600* file. This can be done using the following command. To do this manually, open an administrator command prompt window (right-click Command Prompt and click Run As Administrator).

```
icacls "%ProgramData%\Sybase Central 6.0.0" /grant everyone:F
```

The provider information for SQL Anywhere is created in the repository file using the following commands.

```
scjview.exe -register "C:\Program Files\SQL Anywhere 11\java\sqlanywhere.jpr"
```

The contents of the *sqlanywhere.jpr* file looks like this (some entries have been split across multiple lines for display purposes). The **AdditionalClasspath** lines must be entered on a single line in the *.jpr* file.

```
PluginName=SQL Anywhere 11
PluginId=sqlanywhere1100
PluginClass=ianywhere.sa.plugin.SAPlugin
PluginFile=C:\Program Files\SQL Anywhere 11\java\saplugin.jar
AdditionalClasspath=
    C:\Program Files\SQL Anywhere 11\java\isql.jar;
    C:\Program Files\SQL Anywhere 11\java\salib.jar;
    C:\Program Files\SQL Anywhere 11\java\JComponents1100.jar;
    C:\Program Files\SQL Anywhere 11\java\jlogon.jar;
    C:\Program Files\SQL Anywhere 11\java\debugger.jar;
    C:\Program Files\SQL Anywhere 11\java\jodbc.jar
ClassloaderId=SA1100
InitialLoadOrder=0
```

The *sqlanywhere.jpr* file was created in the *java* folder of the SQL Anywhere installation when you originally installed SQL Anywhere. Use it as the model for the *.jpr* file that you must create as part of the install process. There are also versions of this file for MobiLink, QAnywhere, and UltraLite called *mobilink.jpr*, *qanywhere.jpr*, and *ultralite.jpr* respectively. They are also located in the *java* folder.

Here is a portion of the *.scRepository600* file that was created using the process described above. Some entries have been split across multiple lines for display purposes. In the file, each entry appears on a single line:

```
#  Version: 6.0.0.1154
#  Fri Feb 22 10:22:20 EST 2008
#
SCRepositoryInfo/Version=4
#
Providers/sqlanywhere1100/Version=11.0.0.1297
Providers/sqlanywhere1100/UseClassloader=true
Providers/sqlanywhere1100/ClassloaderId=SA1100
Providers/sqlanywhere1100/Classpath=
    C:\\Program Files\\SQL Anywhere 11\\java\\saplugin.jar
Providers/sqlanywhere1100/Name=SQL Anywhere 11
Providers/sqlanywhere1100/AdditionalClasspath=
    C:\\Program Files\\SQL Anywhere 11\\java\\isql.jar;
    C:\\Program Files\\SQL Anywhere 11\\java\\salib.jar;
    C:\\Program Files\\SQL Anywhere 11\\java\\JComponents1100.jar;
    C:\\Program Files\\SQL Anywhere 11\\java\\jlogon.jar;
```

```
    C:\\Program Files\\SQL Anywhere 11\\java\\debugger.jar;
    C:\\Program Files\\SQL Anywhere 11\\java\\jodbc.jar
Providers/sqlanywhere1100/Provider=ianywhere.sa.plugin.SAPlugin
Providers/sqlanywhere1100/ProviderId=sqlanywhere1100
Providers/sqlanywhere1100/InitialLoadOrder=0
#
```

Notes

- Your installer should create a file similar to this one using the techniques described above. The only changes required are the fully-qualified paths to the JAR files in the Classpath and AdditionalClasspath lines.
- The AdditionalClasspath lines shown above have wrapped to take up additional lines. They must be on a single line in the *.scRepository600* file.
- Backslash characters (\) are represented with an escape sequence of \\ in the *.scRepository600* file.
- The first line indicates the version of the *.scRepository600* file.
- The lines beginning with # are comments.

Step 6: Creating connection profiles for Sybase Central

This step involves the configuration of Sybase Central. If you are not installing Sybase Central, you can skip it.

When Sybase Central is installed on your system, a connection profile for **SQL Anywhere 11 Demo** is created in the *.scRepository600* file. If you do not want to create one or more connection profiles, then you can skip this step.

The following commands were used to create the **SQL Anywhere 11 Demo** connection profile. Use this as a model for creating your own connection profiles.

```
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Name" "SQL Anywhere
11 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/FirstTimeStart"
"false"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Description"
"Suitable Description"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/ProviderId"
"sqlanywhere1100"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Provider" "SQL
Anywhere 11"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileSettings" "DSN\eSQL^0020Anywhere^002011^0020Demo;UID
\eDBA;PWD\e35c624d517fb"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileName" "SQL Anywhere 11 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileType" "SQL Anywhere"
```

The connection profile strings and values can be extracted from the *.scRepository600* file. Define a connection profile using Sybase Central and then look at the *.scRepository600* file for the corresponding lines.

Here is a portion of the *.scRepository600* file that was created using the process described above. Some entries have been split across multiple lines for display purposes. In the file, each entry appears on a single line:

```
#  Version: 6.0.0.1154
#  Fri Feb 23 13:09:14 EST 2007
#
ConnectionProfiles/SQL Anywhere 11 Demo/Name=SQL Anywhere 11 Demo
ConnectionProfiles/SQL Anywhere 11 Demo/FirstTimeStart=false
ConnectionProfiles/SQL Anywhere 11 Demo/Description=Suitable Description
ConnectionProfiles/SQL Anywhere 11 Demo/ProviderId=sqlanywhere1100
ConnectionProfiles/SQL Anywhere 11 Demo/Provider=SQL Anywhere 11
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileSettings=
    DSN\eSQL^0020Anywhere^002011^0020Demo;
    UID\eDBA;
    PWD\e35c624d517fb
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileName=
    SQL Anywhere 11 Demo
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileType=
     SQL Anywhere
```

Step 7: Registering the SQL Anywhere ODBC driver

You must install the SQL Anywhere ODBC driver before it can be used by the iAnywhere JDBC Driver in the administration tools.

For more information, see "Configuring the ODBC driver" on page 902.

Deploying administration tools on Linux/Unix/Mac OS X

This section explains how to install Interactive SQL (dbisql), Sybase Central (including the SQL Anywhere, MobiLink and QAnywhere plug-ins), and the SQL Anywhere Console utility (dbconsole) on Linux, Solaris, and Mac OS X computers. It is intended for those who want to create an installer for these administration tools.

The instructions given here are specific to version 11.0.0 and cannot be applied to earlier or later versions of the software.

Note also that the dbisqlc command line utility is supported on Linux, Solaris, Mac OS X, HP-UX, and AIX. See "Deploying dbisqlc" on page 932.

Check your license agreement
Redistribution of files is subject to your license agreement. No statements in this document override anything in your license agreement. Check your license agreement before considering deployment.

Before you begin

Before you begin, you must install SQL Anywhere on one computer as a source for program files. This is the **reference installation** for your deployment.

The general steps involved are as follows:

1. Decide which programs you want to deploy.
2. Copy the required files.
3. Set environment variables.
4. Register the Sybase Central plug-ins.

Each of these steps is explained in detail in the following sections.

Step 1: Deciding what you want to deploy

You can install any combination of the following software bundles:

- Interactive SQL
- Sybase Central with the SQL Anywhere plug-in
- Sybase Central with the MobiLink plug-in
- Sybase Central with the QAnywhere plug-in
- SQL Anywhere Console utility (dbconsole)

The following components are also required when installing any of the above software bundles:

- The SQL Anywhere ODBC Driver
- The Java Runtime Environment (JRE) version 1.6.0

The instructions in the next section are structured so that you can install any (or all) of these five bundles without conflicts.

Step 2: Copying the required files

Your installer should copy a subset of the files that are installed by the SQL Anywhere installer. You must keep the same directory structure. All files must be installed below the */opt/sqlanywhere11/* directory.

You should preserve the permissions on the files when you copy them from your reference SQL Anywhere installation. In general, all users and groups are allowed to read and execute all files.

The following table lists the files required for each of the software bundles. Make a list of the files you need, and then copy them into the directory structure outlined above. In general, you should take the files from an already-installed copy of SQL Anywhere.

File	Interactive SQL	Sybase Central with the SQL Anywhere plug-in	Sybase Central with the MobiLink plug-in	Sybase Central with the QAnywhere plug-in	SQL Anywhere Console
java/jodbc.jar	X	X	X	X	X

File	Interactive SQL	Sybase Central with the SQL Anywhere plug-in	Sybase Central with the MobiLink plug-in	Sybase Central with the QAnywhere plug-in	SQL Anywhere Console
java/JComponents1100.jar	X	X	X	X	X
java/jlogon.jar	X	X	X	X	X
java/SCEditor600.jar	X	X	X	X	X
java/jsyblib600.jar	X	X	X	X	X
lib32/libjsyblib600_r.so.1	X	X	X	X	X
sun/javahelp-2_0/jh.jar	X	X	X	X	X
jre_1.6.0_linux_sun_i586/... (Linux only)	X	X	X	X	X
jre_1.6.0_solaris_sun_sparc/... (Solaris only)	X	X	X	X	X
lib32/libdblib11_r.so.1	X	X	X	X	X
lib32/libdbjodbc11.so.1	X	X	X	X	X
lib32/libdbodbc11_r.so.1	X	X	X	X	X
lib32/libdbodm11.so.1	X	X	X	X	X
lib32/libdbtasks11_r.so.1	X	X	X	X	X
res/dblg[en]11.res	X	X	X	X	X
lib32/libdbtool11_r.so.1	X	X	X		
bin32/dbisql	X	X			
java/isql.jar	X	X	X		
bin32/scjview		X	X	X	
bin32/scvw[en]600.jar		X	X	X	
java/sybasecentral600.jar		X	X	X	
java/salib.jar		X	X	X	

File	Interactive SQL	Sybase Central with the SQL Anywhere plug-in	Sybase Central with the MobiLink plug-in	Sybase Central with the QAnywhere plug-in	SQL Anywhere Console
java/saplugin.jar		X			
java/debugger.jar		X			
lib32/libdbput11_r.so.1		X			
java/apache_files.txt			X	X	
java/apache_license_1.1.txt			X	X	
java/apache_license_2.0.txt			X	X	
java/log4j.jar			X	X	
java/mlplugin.jar			X		
java/mldesign.jar			X		
java/stax-api-1.0.jar			X		
java/wstx-asl-2.0.5.jar			X		
java/velocity.jar			X		
java/velocity-dep.jar			X		
drivers/...			X		
java/qaplugin.jar				X	
java/qaconnector.jar				X	
java/mlstream.jar				X	
bin32/qaagent				X	
lib32/libdbicu11_r.so				X	
lib32/libdbicudt11.so				X	
bin32/dbinit				X	
bin32/dbconsole					X

File	Interac-tive SQL	Sybase Central with the SQL Any-where plug-in	Sybase Central with the MobiLink plug-in	Sybase Central with the QAny-where plug-in	SQL Any-where Console
java/DBConsole.jar					X

The table above shows a number of files with the designation **[en]**. For Linux, these files are available in a number of different languages with English (**en**) being just one of them. For more information, see the following section "International message and context-sensitive help files" on page 926.

Some file paths above end with "...". This indicates that the entire tree, including subdirectories, should be copied.

Note that on Mac OS X, shared objects have a *.dylib* extension, and symlink (symbolic link) creation is not necessary.

The administration tools require the 32-bit version of JRE 1.6.0. You should not substitute a later patch version of the JRE unless you have a specific need to do so. Not all platform versions of the JRE are bundled with SQL Anywhere. The platforms that are included with SQL Anywhere support Linux on x86/x64 and Solaris SPARC. Other platforms versions must be obtained from the appropriate vendor. For example, if you are working with a Linux install, copy the entire *jre_1.6.0_linux_sun_i586* tree, including subdirectories.

If the platform that you require is included with SQL Anywhere, copy the JRE files from an installed copy of SQL Anywhere 11. Copy the entire tree, including subdirectories.

The *sqlanywhere.jpr* file contains a list of the required component files for the SQL Anywhere plug-in of Sybase Central.

The *mobilink.jpr* file contains a list of the required component files for the MobiLink plug-in of Sybase Central.

The *qanywhere.jpr* file contains a list of the required component files for the QAnywhere plug-in of Sybase Central. When deploying the QAnywhere plug-in, dbinit is required. For information about deploying database tools, see "Deploying database utilities" on page 940.

All five bundles in the table above require a number of links to be created. The following sections provide details.

Base component files

All five bundles require the links listed in this section.

Create the following symbolic links in */opt/sqlanywhere11/lib32*:

```
libdbicu11_r.so -> libdbicu11_r.so.1
libdbicudt11.so -> libdbicudt11.so.1
libdbjodbc11.so -> libdbjodbc11.so.1
libjsyblib600_r.so -> libjsyblib600_r.so.1
libdbodbc11_r.so -> libdbodbc11_r.so.1
libdbodm11.so -> libdbodm11.so.1
libdbtasks11_r.so -> libdbtasks11_r.so.1
```

Create a symbolic link in */opt/sqlanywhere11/sun*. The symbolic link for Linux is *jre160_x86* for the 32-bit JRE. The symbolic link for other systems is *jre_160*.

```
jre160_x86 -> /opt/sqlanywhere11/sun/jre_1.6.0_linux_sun_i586 (Linux)
jre160 -> /opt/sqlanywhere11/sun/jre_1.6.0_solaris_sun_sparc (Solaris)
```

Interactive SQL files

Create the following symbolic links in */opt/sqlanywhere11/lib32*:

```
libdblib11_r.so -> libdblib11_r.so.1
libdbtool11_r.so -> libdbtool11_r.so.1
```

Sybase Central with the SQL Anywhere plug-in

Create the following symbolic links in */opt/sqlanywhere11/lib32*:

```
libdblib11_r.so -> libdblib11_r.so.1
libdbput11_r.so -> libdbput11_r.so.1
libdbtool11_r.so -> libdbtool11_r.so.1
```

Sybase Central with the MobiLink plug-in

Create the following symbolic links in */opt/sqlanywhere11/lib32*:

```
libdblib11_r.so -> libdblib11_r.so.1
libdbmlput11_r.so -> libdbmlput11_r.so.1
libdbtool11_r.so -> libdbtool11_r.so.1
```

Create an additional symbolic link in */opt/sqlanywhere11/sun* for 64-bit Linux. The symbolic link for Linux is *jre160_x64* for the 64-bit JRE.

```
jre160_x64 -> /opt/sqlanywhere11/sun/jre_1.6.0_linux_sun_x64 (Linux)
```

Sybase Central with the QAnywhere plug-in

Create the following symbolic link in */opt/sqlanywhere11/lib32*:

```
libdblib11_r.so -> libdblib11_r.so.1
```

Create an additional symbolic link in */opt/sqlanywhere11/sun* for 64-bit Linux. The symbolic link for Linux is *jre160_x64* for the 64-bit JRE.

```
jre160_x64 -> /opt/sqlanywhere11/sun/jre_1.6.0_linux_sun_x64 (Linux)
```

The SQL Anywhere Console

Create the following symbolic link in */opt/sqlanywhere11/lib32*:

```
libdblib11_r.so -> libdblib11_r.so.1
```

International message and context-sensitive help files

For Linux systems only, all displayed text and context-sensitive help for the administration tools have been translated from English into German, Japanese, and Simplified Chinese. The resources for each language are in separate files. The English files contain **en** in the file names. German file names contain **de**, Japanese file names contain **ja**, and Chinese files contain **zh**.

If you want to install support for different languages, you have to add the message files for those other languages. The translated files are as follows:

`dblgen11.res`	English
`dblgde11_iso_1.res` `dblgde11_utf8.res`	German (Linux only)

`dblgja11_eucjis.res`
`dblgja11_sjis.res`
`dblgja11_utf8.res`
	Japanese (Linux only)

`dblgzh11_cp936.res` `dblgzh11_eucgb.res`	Simplified Chinese (Linux only)

You must also add the context-sensitive help files for those other languages. The available translated files are as follows:

scvwen600.jar	English
scvwde600.jar	German
scvwfr600.jar	French
scvwja600.jar	Japanese
scvwzh600.jar	Simplified Chinese

These files are included with localized versions of SQL Anywhere.

Step 3: Setting environment variables

To run the administration tools, a number of environment variables must be defined or modified. This is usually done in the *sa_config.sh* file, which is created by the SQL Anywhere installer, but you have the flexibility to do this however is most appropriate for your application.

1. Set the PATH to include the following:

   ```
   /opt/sqlanywhere11/bin32
   ```

   or

   ```
   /opt/sqlanywhere11/bin32s
   ```

   (whichever is appropriate).

2. Set LD_LIBRARY_PATH to include the following:

   ```
   /opt/sqlanywhere11/jre160/lib/i386/client
   /opt/sqlanywhere11/jre160/lib/i386
   /opt/sqlanywhere11/jre160/lib/i386/native_threads
   ```

3. Set the following environment variable:

   ```
   SQLANY11="/opt/sqlanywhere11"
   ```

Step 4: Registering the Sybase Central plug-ins

This step involves the configuration of Sybase Central. If you are not installing Sybase Central, you can skip it.

Sybase Central requires a configuration file that lists the installed plug-ins. Your installer must create this file. Note that it contains full paths to a number of JAR files that may change depending on where the software is installed.

The file is called *.scRepository600*. It resides in the */opt/sqlanywhere11/sybcentral600* directory. It is a plain text file that contains some basic information about the plug-ins that Sybase Central should load.

The provider information for SQL Anywhere is created in the repository file using the following commands.

```
scjview -register "/opt/sa11/java/sqlanywhere.jpr"
```

The contents of the *sqlanywhere.jpr* file looks like this (some entries have been split across multiple lines for display purposes). The **AdditionalClasspath** lines must be entered on a single line in the *.jpr* file.

```
PluginName=SQL Anywhere 11
PluginId=sqlanywhere1100
PluginClass=ianywhere.sa.plugin.SAPlugin
PluginFile=\_opt\_sa11\_java\_saplugin.jar
AdditionalClasspath=\_opt\_sa11\_java\_isql.jar:
    \_opt\_sa11\_java\_salib.jar:
    \_opt\_sa11\_java\_JComponents1100.jar:
    \_opt\_sa11\_java\_jlogon.jar:
    \_opt\_sa11\_java\_debugger.jar:
    \_opt\_sa11\_java\_jodbc.jar
ClassloaderId=SA1100
```

The *sqlanywhere.jpr* file was created in the *java* folder of the SQL Anywhere installation when you originally installed SQL Anywhere. Use it as the model for the *.jpr* file that you must create as part of the install process. There are also versions of this file for MobiLink and QAnywhere called *mobilink.jpr* and *qanywhere.jpr* respectively. They are also located in the *java* folder.

Here is a sample *.scRepository600* file that was created using the process described above. Some entries have been split across multiple lines for display purposes. In the file, each entry appears on a single line:

```
#  Version: 6.0.0.1154
#  Fri Feb 23 13:09:14 EST 2007
#
SCRepositoryInfo/Version=4
#
Providers/sqlanywhere1100/Version=11.0.0.1297
Providers/sqlanywhere1100/UseClassloader=true
Providers/sqlanywhere1100/ClassloaderId=SA1100
Providers/sqlanywhere1100/Classpath=
    \_opt\_sa11\_java\_saplugin.jar
Providers/sqlanywhere1100/Name=SQL Anywhere 11
Providers/sqlanywhere1100/AdditionalClasspath=
    \_opt\_sa11\_java\_isql.jar:
    \_opt\_sa11\_java\_salib.jar:
    \_opt\_sa11\_java\_JComponents1100.jar:
    \_opt\_sa11\_java\_jlogon.jar:
    \_opt\_sa11\_java\_debugger.jar:
    \_opt\_sa11\_java\_jodbc.jar
Providers/sqlanywhere1100/Provider=ianywhere.sa.plugin.SAPlugin
```



```
Providers/sqlanywhere1100/ProviderId=sqlanywhere1100
Providers/sqlanywhere1100/InitialLoadOrder=0
#
```

Notes

- Your installer should create a file similar to this one using the techniques described above. The only changes required are the fully-qualified paths to the JAR files in the Classpath and AdditionalClasspath lines.
- The AdditionalClasspath lines shown above have wrapped to take up additional lines. They must be on a single line in the *.scRepository600* file.
- Forward slash characters (/) are represented with an escape sequence of _ in the *.scRepository600* file.
- The first line indicates the version of the *.scRepository600* file.
- The lines beginning with # are comments.

For more information about deploying databases and database applications, see "Deploying databases and applications" on page 881.

Step 5: Creating connection profiles for Sybase Central

This step involves the configuration of Sybase Central. If you are not installing Sybase Central, you can skip it.

When Sybase Central is installed on your system, a connection profile for **SQL Anywhere 11 Demo** is created in the *.scRepository600* file. If you do not want to create one or more connection profiles, then you can skip this step.

The following commands were used to create the **SQL Anywhere 11 Demo** connection profile. Use this as a model for creating your own connection profiles.

```
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Name" "SQL Anywhere
11 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/FirstTimeStart"
"false"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Description"
"Suitable Description"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/ProviderId"
"sqlanywhere1100"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Provider" "SQL
Anywhere 11"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileSettings" "DSN\eSQL^0020Anywhere^002011^0020Demo;UID
\eDBA;PWD\e35c624d517fb"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileName" "SQL Anywhere 11 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileType" "SQL Anywhere"
```

The connection profile strings and values can be extracted from the *.scRepository600* file. Define a connection profile using Sybase Central and then look at the *.scRepository600* file for the corresponding lines.

Here is a portion of the *.scRepository600* file that was created using the process described above. Some entries have been split across multiple lines for display purposes. In the file, each entry appears on a single line:

```
#  Version: 6.0.0.1154
#  Fri Feb 23 13:09:14 EST 2007
#
ConnectionProfiles/SQL Anywhere 11 Demo/Name=SQL Anywhere 11 Demo
ConnectionProfiles/SQL Anywhere 11 Demo/FirstTimeStart=false
ConnectionProfiles/SQL Anywhere 11 Demo/Description=Suitable Description
ConnectionProfiles/SQL Anywhere 11 Demo/ProviderId=sqlanywhere1100
ConnectionProfiles/SQL Anywhere 11 Demo/Provider=SQL Anywhere 11
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileSettings=
    DSN\eSQL^0020Anywhere^002011^0020Demo
    UID\eDBA;
    PWD\e35c624d517fb
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileName=
    SQL Anywhere 11 Demo
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileType=
     SQL Anywhere
```

Configuring the administration tools

You can control which features are shown or enabled by the administration tools. Control is done through an initialization file named *OEM.ini*. This file must be in the same directory as the JAR files used by the administration tools (for example, *C:\Program Files\SQL Anywhere 11\java*). If the file is not found, default values are used. Also, defaults are used for values that are missing from *OEM.ini*.

Here is a sample *OEM.ini* file:

```
[errors]
# reportErrors type is boolean, default = true
reportErrors=false

[updates]
# checkForUpdates type is boolean, default = true
checkForUpdates=false

[dbisql]
disableExecuteAll=true
# lockedPreferences is assigned a comma-separated
# list of one or more of the following option names:
#    autoCommit
#    autoRefetch
#    commitOnExit
#    disableResultsEditing
#    executeToolbarButtonSemantics
#    fastLauncherEnabled
#    maximumDisplayedRows
#    showMultipleResultSets
#    showResultsForAllStatements
lockedPreferences=showMultipleResultSets,commitOnExit
```

Any line beginning with the # character is a comment line and is ignored. The specified option names and values are case-sensitive.

If **reportErrors** is false, the administration tool does not present a window to the user inviting them to submit error information to iAnywhere if the software crashes. Instead, the traditional, simpler window appears.

If **checkForUpdates** is false, the administration tool does not check for SQL Anywhere software updates automatically, nor does it give the user the option to do it at their discretion.

If **disableExecuteAll** is true, then the **SQL** » **Execute** menu item and the F5 accelerator key are disabled in Interactive SQL. If the Execute toolbar button is configured for **Execute All Statement(s)**, then it is disabled also. Therefore, you might want to set the Execute toolbar button to **Execute Selected Statement(s)** in Interactive SQL, and then set the executeToolbarButtonSemantics option in the *OEM.ini* file to prevent users from changing the Execute toolbar button. See "Configuring the Execute Statements toolbar button" [*SQL Anywhere Server - Database Administration*].

Preventing users from changing Interactive SQL option settings

In the **[dbisql]** section of the *OEM.ini* file, you can lock the settings of Interactive SQL options so that users cannot change them. The option names are case sensitive. For some options, you can specify whether the setting is locked for SQL Anywhere databases or UltraLite databases. If you do not specify the database type, then the setting is locked for all databases. The following is an example:

```
[dbisql]
lockedPreferences=autoCommit
```

To lock the option setting for only one type of database, prefix **lockedPreferences** with the name of the database type (**SQLAnywhere** or **UltraLite**) followed by a period.

For example, if you want to lock **autoCommit** for SQL Anywhere databases, but not UltraLite, you would add the following line:

```
[dbisql]
SQLAnywhere.lockedPreferences=autoCommit
```

You can prevent users from changing the following Interactive SQL option settings (**SQLAnywhere/UltraLite** indicates that these options can be locked for a specific database type):

- **autoCommit (SQLAnywhere/UltraLite)** Prevents the user from customizing the **Commit After Every Statement** option. See "auto_commit option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].
- **autoRefetch (SQLAnywhere/UltraLite)** Prevents the user from customizing the **Automatically Refetch Results** option. See "auto_refetch option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].
- **commitOnExit (SQLAnywhere/UltraLite)** Prevents the user from customizing the **Commit On Exit Or Disconnect** option. See "commit_on_exit option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].
- **disableResultsEditing (SQLAnywhere/UltraLite)** Prevents the user from customizing the **Disable Editing** option.
- **executeToolbarButtonSemantics** Prevents the user from customizing the behavior of the **Execute** toolbar button. See "Configuring the Execute Statements toolbar button" [*SQL Anywhere Server - Database Administration*].
- **fastLauncherEnabled** Prevents the user from customizing the fast launcher option.

- **maximumDisplayedRows (SQLAnywhere/UltraLite)** Prevents the user from customizing the **Maximum Number Of Rows To Display** option. "isql_maximum_displayed_rows option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].
- **showMultipleResultSets (SQLAnywhere/UltraLite)** Prevents the user from customizing the **Show Only The First Result Set** or **Show All Result Sets** options. See "isql_show_multiple_result_sets [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].
- **showResultsForAllStatements (SQLAnywhere/UltraLite)** Prevents the user from customizing the **Show Results From The Last Statement** or **Show Results From Each Statement** options.

Deploying dbisqlc

If your customer application is running on computers with limited resources, you may want to deploy the dbisqlc executable instead of Interactive SQL (dbisql). However, you should note that dbisqlc does not contain all the features of Interactive SQL and compatibility between the two is not guaranteed.

The dbisqlc executable requires the standard embedded SQL client-side libraries.

For more information about dbisqlc, see "Interactive SQL utility (dbisqlc)" [*SQL Anywhere Server - Database Administration*].

Deploying SQL script files

The following files are required to unload a SQL Anywhere 11 database.

```
optdeflt.sql
opttemp.sql
```

The following files are required to unload databases built with earlier versions of SQL Anywhere.

```
unloadold.sql
```


Deploying database servers

You can deploy a database server by making the SQL Anywhere installer available to your end users. By selecting the proper option, each end user is guaranteed of getting the files they need.

The simplest way to deploy a personal database server or a network database server is to use the **Deployment Wizard**. For more information, see "Using the Deployment Wizard" on page 887.

To run a database server, you need to install a set of files. The files are listed in the following table. All redistribution of these files is governed by the terms of your license agreement. You must confirm whether you have the right to redistribute the database server files before doing so.

Windows	Linux/Unix	Mac OS X
dbeng11.exe	*dbeng11*	*dbeng11*
dbeng11.lic	*dbeng11.lic*	*dbeng11.lic*
dbsrv11.exe	*dbsrv11*	*dbsrv11*
dbsrv11.lic	*dbsrv11.lic*	*dbsrv11.lic*
dbserv11.dll	*libdbserv11_r.so, libdb-tasks11_r.so*	*libdbserv11_r.dylib, libdb-tasks11_r.dylib*
dbscript11.dll	*libdbscript11_r.so*	*libdbscript11_r.dylib*
dblg[en]11.dll	*dblg[en]11.res*	*dblg[en]11.res*
dbctrs11.dll	N/A	N/A
dbextf.dll [1]	*libdbextf.so* [1]	*libdbextf.dylib* [1]
dbicu11.dll[2]	*libdbicu11_r.so*[2]	*libdbicu11_r.dylib*[2]
dbicudt11.dll[2] [3]	*libdbicudt11.so*[2]	*libdbicudt11.dylib*[2]
sqlany.cvf	*sqlany.cvf*	*sqlany.cvf*
dbodbc11.dll [4]	*libdbodbc11.so* [4]	*libdbodbc11.dylib* [4]
N/A	*libdbodbc11_n.so* [4]	*libdbodbc11_n.dylib* [4]
N/A	*libdbodbc11_r.so* [4]	*libdbodbc11_r.dylib* [4]
dbjodbc11.dll [4]	*libdbjodbc11.so* [4]	*libdbjodbc11.dylib* [4]
java\jconn3.jar [4]	*java/jconn3.jar* [4]	*java/jconn3.jar* [4]

Windows	Linux/Unix	Mac OS X
java\jodbc.jar [4]	*java/jodbc.jar* [4]	*java/jodbc.jar* [4]
java\sajvm.jar [4]	*java/sajvm.jar* [4]	*java/sajvm.jar* [4]
java\cis.zip [5]	*java/cis.zip* [5]	*java/cis.zip* [5]

[1] Required only if using system extended stored procedures and functions (xp_*).

[2] Required only if the database character set is multi-byte or if the UCA collation sequence is used.

[3] On Windows Mobile, the file name to deploy is called *dbicudt11.dat*.

[4] Required only if using Java in the database.

[5] Required only if using Java in the database and remote data access.

Notes

- Depending on your situation, you should choose whether to deploy the personal database server (dbeng11) or the network database server (dbsrv11).
- You must include the separate corresponding license file (*dbeng11.lic* or *dbsrv11.lic*) when deploying a database server. The license files are located in the same directory as the server executables.
- The Java VM jar file (*sajvm.jar*) is required only if the database server is to use the Java in the Database functionality.
- The table does not include files needed to run utilities such as dbbackup.

  For information about deploying utilities, see "Deploying administration tools" on page 910.

Windows Registry entries

To ensure that messages written by the server to the Event Log on Windows are formatted correctly, create the following registry key.

```
HKEY_LOCAL_MACHINE\
   SYSTEM\
      CurrentControlSet\
         Services\
            Eventlog\
               Application\
                  SQLANY 11.0
```

Within this key, add a REG_SZ value named EventMessageFile and assign it the data value of the fully qualified location of *dblgen11.dll*, for example, *C:\Program Files\SQL Anywhere 11\bin32\dblgen11.dll*. The English language DLL, *dblgen11.dll*, can be specified regardless of the deployment language. Here is a sample registry change file.

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application
\SQLANY 11.0]
"EventMessageFile"="c:\\sa11\\bin32\\dblgen11.dll"
```

For the 64-bit version of the server, the registry key is `SQLANY64 11.0`.

To ensure that messages written by MESSAGE ... TO EVENT LOG statements to the Event Log on Windows are formatted correctly, create the following registry key.

```
HKEY_LOCAL_MACHINE\
   SYSTEM\
      CurrentControlSet\
         Services\
            Eventlog\
               Application\
                  SQLANY 11.0 Admin
```

Within this key, add a REG_SZ value named EventMessageFile and assign it the data value of the fully qualified location of *dblgen11.dll*, for example, *C:\Program Files\SQL Anywhere 11\bin32\dblgen11.dll*. The English language DLL, *dblgen11.dll*, can be specified regardless of the deployment language. Here is a sample registry change file.

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application
\SQLANY 11.0 Admin]
"EventMessageFile"="c:\\sa11\\bin32\\dblgen11.dll"
```

For the 64-bit version of the server, the registry key is `SQLANY64 Admin 11.0`.

You can suppress Windows event log entries by setting up a registry key. The registry key is:

```
Software\Sybase\SQL Anywhere\11.0\EventLogMask
```

and it can be placed in either the HKEY_CURRENT_USER or HKEY_LOCAL_MACHINE hive. To control event log entries, create a REG_DWORD value named EventLogMask and assign it a bit mask containing the internal bit values for the different Windows event types. The three types supported by the SQL Anywhere database server are:

```
EVENTLOG_ERROR_TYPE        0x0001
EVENTLOG_WARNING_TYPE      0x0002
EVENTLOG_INFORMATION_TYPE  0x0004
```

For example, if the EventLogMask key is set to zero, no messages appear at all. A better setting would be 1, so that informational and warning messages don't appear, but errors do. The default setting (no entry present) is for all message types to appear. Here is a sample registry change file.

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\SQL Anywhere\11.0]
"EventLogMask"=dword:00000007
```

Registering DLLs on Windows

When deploying SQL Anywhere, there are DLL files that must be registered for SQL Anywhere to function properly. Note that for Windows Vista or later versions of Windows, you must include the SQL Anywhere elevated operations agent (*dbelevate11.exe*) which supports the privilege elevation required when DLLs are registered or unregistered.

There are many ways you can register these DLLs, including in an install script or using the regsvr32 utility on Windows or the regsvrce utility on Windows Mobile. You could also include a command, like the one in the following procedure, in a batch file.

To register a DLL

1. Open a command prompt.
2. Change to the directory where the DLL provider is installed.
3. Enter the following command to register the provider (in this example, the OLE DB provider is registered):

   ```
   regsvr32 dboledb11.dll
   ```

The following table lists the DLLs that must be registered when deploying SQL Anywhere:

File	Description
dbctrs11.dll	The SQL Anywhere Performance Monitor counters.
dbmlsynccom.dll	The Dbmlsync Integration Component (non-visual component).
dbmlsynccomg.dll	The Dbmlsync Integration Component (visual component).
dbodbc11.dll	The SQL Anywhere ODBC driver.
dboledb11.dll	The SQL Anywhere OLE DB provider.
dboledba11.dll	The SQL Anywhere OLE DB provider schema assist module.
Windows\system32\msxml4.dll	The Microsoft XML Parser.

Deploying databases

You deploy a database file by installing the database file onto your end-user's disk.

As long as the database server shuts down cleanly, you do not need to deploy a transaction log file with your database file. When your end user starts running the database, a new transaction log is created.

For SQL Remote applications, the database should be created in a properly synchronized state, in which case no transaction log is needed. You can use the Extraction utility for this purpose.

For information about extracting databases, see "Database Extraction utility" [*SQL Remote*].

International considerations

When you are deploying a database worldwide, you should consider the locales in which the database is to be used. Different locales may have different sort orders or text comparison rules. For example, the database that you are going to deploy maybe have been created using the 1252LATIN1 collation and this may not be suitable for some of the environments in which it is to be used.

Since the collation of a database cannot be changed after it has been created, you might consider creating the database during the install phase and then populating the database with the schema and data that you

want it to have. The database can be created during the install either by using the dbinit utility, or by starting the database server with the utility database and issuing a CREATE DATABASE statement. Then, you can use SQL statements to create the schema and do whatever else is necessary to set up your initial database.

If you decide to use the UCA collation, you can specify additional collation tailoring options for finer control over the sorting and comparing of characters using the dbinit utility or the CREATE DATABASE statement. These options take the form of *keyword=value* pairs, assembled in parentheses, following the collation name. Using the CREATE DATABASE statement, for example, a collation tailoring can be specified using syntax like the following:

```
CHAR COLLATION 'UCA( locale=es;case=respect;accent=respect )'
```

As an alternative, you could create multiple database templates, one for each of the locales in which the database is to be used. This may be suitable if the set of locales you are deploying the database to is relatively small. You can have the installer choose which database to install.

See also

- "CREATE DATABASE statement" [*SQL Anywhere Server - SQL Reference*]
- "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*]
- "Choosing collations" [*SQL Anywhere Server - Database Administration*]

Deploying databases on read-only media

You can distribute databases on read-only media, such as a CD-ROM, as long as you run them in read-only mode.

For more information about running databases in read-only mode, see "-r server option" [*SQL Anywhere Server - Database Administration*].

If you need to make changes to the database, you must copy the database from the CD-ROM to a location where it can be modified, such as a hard drive.

Deploying security

The following table summarizes the components that support security features in SQL Anywhere.

Security option	Security type	Included in module	Licensable
Database encryption	AES	*dbserv11.dll* *libdbserv11_r.so*	included[1]
Database encryption	FIPS-approved AES	*dbfips11.dll*	separately licensed[2]
Transport layer security	RSA	*dbrsa11.dll* *libdbrsa11.so* *libdbrsa11.dylib* *libdbrsa11_r.dylib*	included[1]
Transport layer security	FIPS-approved RSA	*dbfips11.dll, sbgse2.dll*	separately licensed[2]
Transport layer security	ECC	*dbecc11.dll* *libdbecc11.so*	separately licensed[2]

[1] AES and RSA strong encryption are included with SQL Anywhere and do not require a separate license, but these libraries are not FIPS-certified.

[2] The software for strong encryption using ECC or FIPS-certified technology must be ordered separately.

Deploying embedded database applications

This section provides information on deploying embedded database applications, where the application and the database both reside on the same computer.

An embedded database application includes the following:

- **Client application** This includes the SQL Anywhere client requirements.

  For information about deploying client applications, see "Deploying client applications" on page 891.
- **Database server** The SQL Anywhere personal database server.

  For information about deploying database servers, see "Deploying database servers" on page 934.
- **SQL Remote** If your application uses SQL Remote replication, you must deploy the SQL Remote Message Agent.
- **The database** You must deploy a database file holding the data the application uses.

Deploying personal servers

When you deploy an application that uses the personal server, you need to deploy both the client application components and the database server components.

The language resource library (*dblgen11.dll*) is shared between the client and the server. You need only one copy of this file.

It is recommended that you follow the SQL Anywhere installation behavior, and install the client and server files in the same directory.

Remember to provide the Java zip files and the Java DLL if your application takes advantage of Java in the Database.

Deploying database utilities

If you need to deploy database utilities (such as dbbackup) along with your application, then you need the utility executable together with the following additional files:

Description	Windows	Linux/Unix	Mac OS X
Database tools library	*dbtool11.dll*	*libdbtool11.so, libdb-tasks11.so*	*libdbtool11.dylib, libdb-tasks11.dylib*
Interface Library	*dblib11.dll*	*libdblib11.so*	*libdblib11.dylib*
Language resource library	*dblg[en]11.dll*	*dblg[en]11.res*	*dblg[en]11.res*
Connect window	*dbcon11.dll*		

Description	Windows	Linux/Unix	Mac OS X
Pre-10 physical store library	*dboftsp.dll*	*libdboftsp.so*	*libdboftsp.dylib*

Notes

- For multi-threaded applications on Linux/Unix, use *libdbtool11_r.so*, *libdbtasks11_r.so*, and *libdblib11_r.so*.
- For multi-threaded applications on Mac OS X, use *libdbtool11_r.dylib*, *libdbtasks11_r.dylib*, and *libdblib11_r.dylib*.
- The pre-10 physical store library is required by some utilities (dblog, dbtran, dberase) in order to access pre-version 10.0.0 log files. If you are not deploying these utilities, then you do not require this library.
- The personal database server (dbeng11) is required for creating databases using the dbinit utility. It is also required if you are creating databases from Sybase Central on the local computer when no other database servers are running. See "Deploying database servers" on page 934.
- The dbunload utility may require the contents of the *scripts* directory to be present. See "Deploying SQL script files" on page 933.

Deploying unload support for pre 10.0.0 databases

If, in your application, you need to the ability to convert older databases to the 11.0.0 format, then you need the database unload utility, dbunload, together with the following additional files:

Description	Windows	Linux/Unix	Mac OS X
Unload support for pre-10.0 databases	*dbunlspt.exe*	*dbunlspt*	*dbunlspt*
Message resource library	*dbus[en].dll*	*dbus[en].res*	*dbus[en].res*

The table above shows a file with the designation **[en]**. These files are available in a number of different languages with English (**en**) being just one of them. For more information, see the following section "International message files" on page 941.

In addition to these files, you also need the files described in "Deploying database utilities" on page 940 and "Deploying SQL script files" on page 933.

International message files

All displayed text for the pre-10.0.0 unload support tool is translated from English into French, German, Japanese, and Simplified Chinese. The resources for each language are held in separate files. The English file contains **en** in the file name. The French file has a similar name, but uses **fr** instead of **en**. The German file name contains **de**, the Japanese file name contains **ja**, and the Chinese file name contains **zh**.

If you want to install support for different languages, you have to add the resource files for those other languages. The translated files are as follows.

Windows Message Files (*.dll* files):

dbusen.dll	English
dbusde.dll	German
dbuses.dll	Spanish
dbusfr.dll	French
dbusit.dll	Italian
dbusja.dll	Japanese
dbusko.dll	Korean
dbuslt.dll	Lithuanian
dbuspl.dll	Polish
dbuspt.dll	Portuguese
dbusru.dll	Russian
dbustw.dll	Traditional Chinese
dbusuk.dll	Ukrainian
dbuszh.dll	Simplified Chinese

Linux/Unix/Mac OS X Message Files (*.res* files):

dbusen.res	English
dbusfr.res	French
dbusde.res	German
dbusja.res	Japanese
dbuszh.res	Chinese

These files are included with localized versions of SQL Anywhere.

Deploying SQL Remote

If you are deploying the SQL Remote Message Agent, you need to include the following files:

Description	Windows	Linux/Unix
Message Agent	*dbremote.exe*	*dbremote*
Database tools library	*dbtool11.dll*	*libdbtool11.so, libdbtasks11.so*
Encryption/decryption module	*dbencod11.dll*	*libdbencod11_r.so*
Language resource library	*dblg[en]11.dll*	*dblg[en]11.res*
SMTP message link library [1]	*dbsmtp11.dll*	
FILE message link library [1]	*dbfile11.dll*	*libdbfile11.so*
FTP message link library [1]	*dbftp11.dll*	
Interface Library	*dblib11.dll*	

[1] Only deploy the library for the message link you are using.

It is recommended that you follow the SQL Anywhere installation behavior, and install the SQL Remote files in the same directory as the SQL Anywhere files.

For multi-threaded applications on Linux/Unix, use *libdbtool11_r.so* and *libdbtasks11_r.so*.

Part VI. Glossary

CHAPTER 25

Glossary

Adaptive Server Anywhere (ASA)

The relational database server component of SQL Anywhere Studio, intended for use in mobile and embedded environments or as a server for small and medium-sized businesses. In version 10.0.0, Adaptive Server Anywhere was renamed SQL Anywhere Server, and SQL Anywhere Studio was renamed SQL Anywhere.

See also: "SQL Anywhere" on page 970.

agent ID

See also: "client message store ID" on page 949.

article

In MobiLink or SQL Remote, an article is a database object that represents a whole table, or a subset of the columns and rows in a table. Articles are grouped together in a publication.

See also:

- "replication" on page 968
- "publication" on page 965

atomic transaction

A transaction that is guaranteed to complete successfully or not at all. If an error prevents part of an atomic transaction from completing, the transaction is rolled back to prevent the database from being left in an inconsistent state.

base table

Permanent tables for data. Tables are sometimes called **base tables** to distinguish them from temporary tables and views.

See also:

- "temporary table" on page 972
- "view" on page 974

bit array

A bit array is a type of array data structure that is used for efficient storage of a sequence of bits. A bit array is similar to a character string, except that the individual pieces are 0s (zeros) and 1s (ones) instead of characters. Bit arrays are typically used to hold a string of Boolean values.

business rule

A guideline based on real-world requirements. Business rules are typically implemented through check constraints, user-defined data types, and the appropriate use of transactions.

See also:

- "constraint" on page 950
- "user-defined data type" on page 974

carrier

A MobiLink object, stored in MobiLink system tables or a Notifier properties file, that contains information about a public carrier for use by server-initiated synchronization.

See also: "server-initiated synchronization" on page 970.

character set

A character set is a set of symbols, including letters, digits, spaces, and other symbols. An example of a character set is ISO-8859-1, also known as Latin1.

See also:

- "code page" on page 949
- "encoding" on page 954
- "collation" on page 949

check constraint

A restriction that enforces specified conditions on a column or set of columns.

See also:

- "constraint" on page 950
- "foreign key constraint" on page 956
- "primary key constraint" on page 965
- "unique constraint" on page 973

checkpoint

The point at which all changes to the database are saved to the database file. At other times, committed changes are saved only to the transaction log.

checksum

The calculated number of bits of a database page that is recorded with the database page itself. The checksum allows the database management system to validate the integrity of the page by ensuring that the numbers match as the page is being written to disk. If the counts match, it's assumed that page was successfully written.

client message store

In QAnywhere, a SQL Anywhere database on the remote device that stores messages.

client message store ID

In QAnywhere, a MobiLink remote ID that uniquely identifies a client message store.

client/server

A software architecture where one application (the client) obtains information from and sends information to another application (the server). The two applications often reside on different computers connected by a network.

code page

A code page is an encoding that maps characters of a character set to numeric representations, typically an integer between 0 and 255. An example of a code page is Windows code page 1252. For the purposes of this documentation, code page and encoding are interchangeable terms.

See also:

- “character set” on page 948
- “encoding” on page 954
- “collation” on page 949

collation

A combination of a character set and a sort order that defines the properties of text in the database. For SQL Anywhere databases, the default collation is determined by the operating system and language on which the server is running; for example, the default collation on English Windows systems is 1252LATIN1. A collation, also called a collating sequence, is used for comparing and sorting strings.

See also:

- “character set” on page 948
- “code page” on page 949
- “encoding” on page 954

command file

A text file containing SQL statements. Command files can be built manually, or they can be built automatically by database utilities. The dbunload utility, for example, creates a command file consisting of the SQL statements necessary to recreate a given database.

communication stream

In MobiLink, the network protocol used for communication between the MobiLink client and the MobiLink server.

concurrency

The simultaneous execution of two or more independent, and possibly competing, processes. SQL Anywhere automatically uses locking to isolate transactions and ensure that each concurrent application sees a consistent set of data.

See also:

- "transaction" on page 972
- "isolation level" on page 958

conflict resolution

In MobiLink, conflict resolution is logic that specifies what to do when two users modify the same row on different remote databases.

connection ID

A unique number that identifies a given connection between a client application and the database. You can determine the current connection ID using the following SQL statement:

```
SELECT CONNECTION_PROPERTY( 'Number' );
```

connection-initiated synchronization

A form of MobiLink server-initiated synchronization in which synchronization is initiated when there are changes to connectivity.

See also: "server-initiated synchronization" on page 970.

connection profile

A set of parameters that are required to connect to a database, such as user name, password, and server name, that is stored and used as a convenience.

consolidated database

In distributed database environments, a database that stores the master copy of the data. In case of conflict or discrepancy, the consolidated database is considered to have the primary copy of the data.

See also:

- "synchronization" on page 972
- "replication" on page 968

constraint

A restriction on the values contained in a particular database object, such as a table or column. For example, a column may have a uniqueness constraint, which requires that all values in the column be different. A table may have a foreign key constraint, which specifies how the information in the table relates to data in some other table.

See also:

- "check constraint" on page 948
- "foreign key constraint" on page 956
- "primary key constraint" on page 965
- "unique constraint" on page 973

contention

The act of competing for resources. For example, in database terms, two or more users trying to edit the same row of a database contend for the rights to edit that row.

correlation name

The name of a table or view that is used in the FROM clause of a query—either its original name, or an alternate name, that is defined in the FROM clause.

creator ID

In UltraLite Palm OS applications, an ID that is assigned when the application is created.

cursor

A named linkage to a result set, used to access and update rows from a programming interface. In SQL Anywhere, cursors support forward and backward movement through the query results. Cursors consist of two parts: the cursor result set, typically defined by a SELECT statement; and the cursor position.

See also:

- "cursor result set" on page 951
- "cursor position" on page 951

cursor position

A pointer to one row within the cursor result set.

See also:

- "cursor" on page 951
- "cursor result set" on page 951

cursor result set

The set of rows resulting from a query that is associated with a cursor.

See also:

- "cursor" on page 951
- "cursor position" on page 951

data cube

A multi-dimensional result set with each dimension reflecting a different way to group and sort the same results. Data cubes provide complex information about data that would otherwise require self-join queries and correlated subqueries. Data cubes are a part of OLAP functionality.

data definition language (DDL)

The subset of SQL statements for defining the structure of data in the database. DDL statements create, modify, and remove database objects, such as tables and users.

data manipulation language (DML)

The subset of SQL statements for manipulating data in the database. DML statements retrieve, insert, update, and delete data in the database.

data type

The format of data, such as CHAR or NUMERIC. In the ANSI SQL standard, data types can also include a restriction on size, character set, and collation.

See also: "domain" on page 954.

database

A collection of tables that are related by primary and foreign keys. The tables hold the information in the database. The tables and keys together define the structure of the database. A database management system accesses this information.

See also:

- "foreign key" on page 955
- "primary key" on page 965
- "database management system (DBMS)" on page 952
- "relational database management system (RDBMS)" on page 967

database administrator (DBA)

The user with the permissions required to maintain the database. The DBA is generally responsible for all changes to a database schema, and for managing users and groups. The role of database administrator is automatically built into databases as user ID DBA with password sql.

database connection

A communication channel between a client application and the database. A valid user ID and password are required to establish a connection. The privileges granted to the user ID determine the actions that can be carried out during the connection.

database file

A database is held in one or more database files. There is an initial file, and subsequent files are called dbspaces. Each table, including its indexes, must be contained within a single database file.

See also: "dbspace" on page 953.

database management system (DBMS)

A collection of programs that allow you to create and use databases.

See also: "relational database management system (RDBMS)" on page 967.

database name

The name given to a database when it is loaded by a server. The default database name is the root of the initial database file.

See also: "database file" on page 952.

database object

A component of a database that contains or receives information. Tables, indexes, views, procedures, and triggers are database objects.

database owner (dbo)

A special user that owns the system objects not owned by SYS.

See also:

- "database administrator (DBA)" on page 952
- "SYS" on page 972

database server

A computer program that regulates all access to information in a database. SQL Anywhere provides two types of servers: network servers and personal servers.

DBA authority

The level of permission that enables a user to do administrative activity in the database. The DBA user has DBA authority by default.

See also: "database administrator (DBA)" on page 952.

dbspace

An additional database file that creates more space for data. A database can be held in up to 13 separate files (an initial file and 12 dbspaces). Each table, together with its indexes, must be contained in a single database file. The SQL command CREATE DBSPACE adds a new file to the database.

See also: "database file" on page 952.

deadlock

A state where a set of transactions arrives at a place where none can proceed.

device tracking

Functionality in MobiLink server-initiated synchronization that allows you to address messages using the MobiLink user name that identifies a remote device.

See also: "server-initiated synchronization" on page 970.

direct row handling

In MobiLink, a way to synchronize table data to sources other than the MobiLink-supported consolidated databases. You can implement both uploads and downloads with direct row handling.

See also:

- "consolidated database" on page 950
- "SQL-based synchronization" on page 970

domain

Aliases for built-in data types, including precision and scale values where applicable, and optionally including DEFAULT values and CHECK conditions. Some domains, such as the monetary data types, are pre-defined in SQL Anywhere. Also called user-defined data type.

See also: "data type" on page 952.

download

The stage in synchronization where data is transferred from the consolidated database to a remote database.

dynamic SQL

SQL that is generated programmatically by your program before it is executed. UltraLite dynamic SQL is a variant designed for small-footprint devices.

EBF

Express Bug Fix. An express bug fix is a subset of the software with one or more bug fixes. The bug fixes are listed in the release notes for the update. Bug fix updates may only be applied to installed software with the same version number. Some testing has been performed on the software, but the software has not undergone full testing. You should not distribute these files with your application unless you have verified the suitability of the software yourself.

embedded SQL

A programming interface for C programs. SQL Anywhere embedded SQL is an implementation of the ANSI and IBM standard.

encoding

Also known as character encoding, an encoding is a method by which each character in a character set is mapped onto one or more bytes of information, typically represented as a hexadecimal number. An example of an encoding is UTF-8.

See also:

- "character set" on page 948
- "code page" on page 949
- "collation" on page 949

event model

In MobiLink, the sequence of events that make up a synchronization, such as begin_synchronization and download_cursor. Events are invoked if a script is created for them.

external login

An alternate login name and password used when communicating with a remote server. By default, SQL Anywhere uses the names and passwords of its clients whenever it connects to a remote server on behalf of those clients. However, this default can be overridden by creating external logins. External logins are alternate login names and passwords used when communicating with a remote server.

extraction

In SQL Remote replication, the act of unloading the appropriate structure and data from the consolidated database. This information is used to initialize the remote database.

See also: "replication" on page 968.

failover

Switching to a redundant or standby server, system, or network on failure or unplanned termination of the active server, system, or network. Failover happens automatically.

FILE

In SQL Remote replication, a message system that uses shared files for exchanging replication messages. This is useful for testing and for installations without an explicit message-transport system.

See also:"replication" on page 968

file-based download

In MobiLink, a way to synchronize data in which downloads are distributed as files, allowing offline distribution of synchronization changes.

file-definition database

In MobiLink, a SQL Anywhere database that is used for creating download files.

See also: "file-based download" on page 955.

foreign key

One or more columns in a table that duplicate the primary key values in another table. Foreign keys establish relationships between tables.

See also:

- "primary key" on page 965
- "foreign table" on page 956

foreign key constraint

A restriction on a column or set of columns that specifies how the data in the table relates to the data in some other table. Imposing a foreign key constraint on a set of columns makes those columns the foreign key.

See also:

- "constraint" on page 950
- "check constraint" on page 948
- "primary key constraint" on page 965
- "unique constraint" on page 973

foreign table

The table containing the foreign key.

See also: "foreign key" on page 955.

full backup

A backup of the entire database, and optionally, the transaction log. A full backup contains all the information in the database and thus provides protection in the event of a system or media failure.

See also: "incremental backup" on page 957.

gateway

A MobiLink object, stored in MobiLink system tables or a Notifier properties file, that contains information about how to send messages for server-initiated synchronization.

See also: "server-initiated synchronization" on page 970.

generated join condition

A restriction on join results that is automatically generated. There are two types: key and natural. Key joins are generated when you specify KEY JOIN or when you specify the keyword JOIN but do not use the keywords CROSS, NATURAL, or ON. For a key join, the generated join condition is based on foreign key relationships between tables. Natural joins are generated when you specify NATURAL JOIN; the generated join condition is based on common column names in the two tables.

See also:

- "join" on page 959
- "join condition" on page 959

generation number

In MobiLink, a mechanism for forcing remote databases to upload data before applying any more download files.

See also: "file-based download" on page 955.

global temporary table

A type of temporary table for which data definitions are visible to all users until explicitly dropped. Global temporary tables let each user open their own identical instance of a table. By default, rows are deleted on commit, and rows are always deleted when the connection is ended.

See also:

- "temporary table" on page 972
- "local temporary table" on page 960

grant option

The level of permission that allows a user to grant permissions to other users.

hash

A hash is an index optimization that transforms index entries into keys. An index hash aims to avoid the expensive operation of finding, loading, and then unpacking the rows to determine the indexed value, by including enough of the actual row data with its row ID.

histogram

The most important component of column statistics, histograms are a representation of data distribution. SQL Anywhere maintains histograms to provide the optimizer with statistical information about the distribution of values in columns.

iAnywhere JDBC driver

The iAnywhere JDBC driver provides a JDBC driver that has some performance benefits and feature benefits compared to the pure Java jConnect JDBC driver, but which is not a pure-Java solution. The iAnywhere JDBC driver is recommended in most cases.

See also:

- "JDBC" on page 959
- "jConnect" on page 959

identifier

A string of characters used to reference a database object, such as a table or column. An identifier may contain any character from A through Z, a through z, 0 through 9, underscore (_), at sign (@), number sign (#), or dollar sign ($).

incremental backup

A backup of the transaction log only, typically used between full backups.

See also: "transaction log" on page 973.

index

A sorted set of keys and pointers associated with one or more columns in a base table. An index on one or more columns of a table can improve performance.

InfoMaker

A reporting and data maintenance tool that lets you create sophisticated forms, reports, graphs, cross-tabs, and tables, as well as applications that use these reports as building blocks.

inner join

A join in which rows appear in the result set only if both tables satisfy the join condition. Inner joins are the default.

See also:

- "join" on page 959
- "outer join" on page 963

integrated login

A login feature that allows the same single user ID and password to be used for operating system logins, network logins, and database connections.

integrity

Adherence to rules that ensure that data is correct and accurate, and that the relational structure of the database is intact.

See also: "referential integrity" on page 967.

Interactive SQL

A SQL Anywhere application that allows you to query and alter data in your database, and modify the structure of your database. Interactive SQL provides a pane for you to enter SQL statements, as well as panes that return information about how the query was processed and the result set.

isolation level

The degree to which operations in one transaction are visible to operations in other concurrent transactions. There are four isolation levels, numbered 0 through 3. Level 3 provides the highest level of isolation. Level 0 is the default setting. SQL Anywhere also supports three snapshot isolation levels: snapshot, statement-snapshot, and readonly-statement-snapshot.

See also: "snapshot isolation" on page 970.

JAR file

Java archive file. A compressed file format consisting of a collection of one or more packages used for Java applications. It includes all the resources necessary to install and run a Java program in a single compressed file.

Java class

The main structural unit of code in Java. It is a collection of procedures and variables grouped together because they all relate to a specific, identifiable category.

jConnect

A Java implementation of the JavaSoft JDBC standard. It provides Java developers with native database access in multi-tier and heterogeneous environments. However, the iAnywhere JDBC driver is the preferred JDBC driver for most cases.

See also:

- "JDBC" on page 959
- "iAnywhere JDBC driver" on page 957

JDBC

Java Database Connectivity. A SQL-language programming interface that allows Java applications to access relational data. The preferred JDBC driver is the iAnywhere JDBC driver.

See also:

- "jConnect" on page 959
- "iAnywhere JDBC driver" on page 957

join

A basic operation in a relational system that links the rows in two or more tables by comparing the values in specified columns.

join condition

A restriction that affects join results. You specify a join condition by inserting an ON clause or WHERE clause immediately after the join. In the case of natural and key joins, SQL Anywhere generates a join condition.

See also:

- "join" on page 959
- "generated join condition" on page 956

join type

SQL Anywhere provides four types of joins: cross join, key join, natural join, and joins using an ON clause.

See also: "join" on page 959.

Listener

A program, dblsn, that is used by MobiLink server-initiated synchronization. Listeners are installed on remote devices and configured to initiate actions on the device when they receive information from a Notifier.

See also: “server-initiated synchronization” on page 970.

local temporary table

A type of temporary table that exists only for the duration of a compound statement or until the end of the connection. Local temporary tables are useful when you need to load a set of data only once. By default, rows are deleted on commit.

See also:

- “temporary table” on page 972
- “global temporary table” on page 957

lock

A concurrency control mechanism that protects the integrity of data during the simultaneous execution of multiple transactions. SQL Anywhere automatically applies locks to prevent two connections from changing the same data at the same time, and to prevent other connections from reading data that is in the process of being changed.

You control locking by setting the isolation level.

See also:

- “isolation level” on page 958
- “concurrency” on page 949
- “integrity” on page 958

log file

A log of transactions maintained by SQL Anywhere. The log file is used to ensure that the database is recoverable in the event of a system or media failure, to improve database performance, and to allow data replication using SQL Remote.

See also:

- “transaction log” on page 973
- “transaction log mirror” on page 973
- “full backup” on page 956

logical index

A reference (pointer) to a physical index. There is no indexing structure stored on disk for a logical index.

LTM

Log Transfer Manager (LTM) also called Replication Agent. Used with Replication Server, the LTM is the program that reads a database transaction log and sends committed changes to Sybase Replication Server.

See: “Replication Server” on page 968.

maintenance release

A maintenance release is a complete set of software that upgrades installed software from an older version with the same major version number (version number format is *major.minor.patch.build*). Bug fixes and other changes are listed in the release notes for the upgrade.

materialized view

A materialized view is a view that has been computed and stored on disk. Materialized views have characteristics of both views (they are defined using a query specification), and of tables (they allow most table operations to be performed on them).

See also:

- "base table" on page 947
- "view" on page 974

message log

A log where messages from an application such as a database server or MobiLink server can be stored. This information can also appear in a messages window or be logged to a file. The message log includes informational messages, errors, warnings, and messages from the MESSAGE statement.

message store

In QAnywhere, databases on the client and server device that store messages.

See also:

- "client message store" on page 948
- "server message store" on page 970

message system

In SQL Remote replication, a protocol for exchanging messages between the consolidated database and a remote database. SQL Anywhere includes support for the following message systems: FILE, FTP, and SMTP.

See also:

- "replication" on page 968
- "FILE" on page 955

message type

In SQL Remote replication, a database object that specifies how remote users communicate with the publisher of a consolidated database. A consolidated database may have several message types defined for it; this allows different remote users to communicate with it using different message systems.

See also:

- "replication" on page 968
- "consolidated database" on page 950

metadata

Data about data. Metadata describes the nature and content of other data.

See also: "schema" on page 969.

mirror log

See also: "transaction log mirror" on page 973.

MobiLink

A session-based synchronization technology designed to synchronize UltraLite and SQL Anywhere remote databases with a consolidated database.

See also:

- "consolidated database" on page 950
- "synchronization" on page 972
- "UltraLite" on page 973

MobiLink client

There are two kinds of MobiLink clients. For SQL Anywhere remote databases, the MobiLink client is the dbmlsync command line utility. For UltraLite remote databases, the MobiLink client is built in to the UltraLite runtime library.

MobiLink Monitor

A graphical tool for monitoring MobiLink synchronizations.

MobiLink server

The computer program that runs MobiLink synchronization, mlsrv11.

MobiLink system table

System tables that are required by MobiLink synchronization. They are installed by MobiLink setup scripts into the MobiLink consolidated database.

MobiLink user

A MobiLink user is used to connect to the MobiLink server. You create the MobiLink user on the remote database and register it in the consolidated database. MobiLink user names are entirely independent of database user names.

network protocol

The type of communication, such as TCP/IP or HTTP.

network server

A database server that accepts connections from computers sharing a common network.

See also: "personal server" on page 964.

normalization

The refinement of a database structure to eliminate redundancy and improve organization according to rules based on relational database theory.

Notifier

A program that is used by MobiLink server-initiated synchronization. Notifiers run on the same computer as the MobiLink server. They poll the consolidated database for push requests, and then send notifications to Listeners.

See also:

- "server-initiated synchronization" on page 970
- "Listener" on page 959

object tree

In Sybase Central, the hierarchy of database objects. The top level of the object tree shows all products that your version of Sybase Central supports. Each product expands to reveal its own sub-tree of objects.

See also: "Sybase Central" on page 972.

ODBC

Open Database Connectivity. A standard Windows interface to database management systems. ODBC is one of several interfaces supported by SQL Anywhere.

ODBC Administrator

A Microsoft program included with Windows operating systems for setting up ODBC data sources.

ODBC data source

A specification of the data a user wants to access via ODBC, and the information needed to get to that data.

outer join

A join that preserves all the rows in a table. SQL Anywhere supports left, right, and full outer joins. A left outer join preserves the rows in the table to the left of the join operator, and returns a null when a row in the right table does not satisfy the join condition. A full outer join preserves all the rows from both tables.

See also:

- "join" on page 959
- "inner join" on page 958

package

In Java, a collection of related classes.

parse tree

An algebraic representation of a query.

PDB

A Palm database file.

performance statistic

A value reflecting the performance of the database system. The CURRREAD statistic, for example, represents the number of file reads issued by the engine that have not yet completed.

personal server

A database server that runs on the same computer as the client application. A personal database server is typically used by a single user on a single computer, but it can support several concurrent connections from that user.

physical index

The actual indexing structure of an index, as it is stored on disk.

plug-in module

In Sybase Central, a way to access and administer a product. Plug-ins are usually installed and registered automatically with Sybase Central when you install the respective product. Typically, a plug-in appears as a top-level container, in the Sybase Central main window, using the name of the product itself; for example, SQL Anywhere.

See also: "Sybase Central" on page 972.

policy

In QAnywhere, the way you specify when message transmission should occur.

polling

In MobiLink server-initiated synchronization, the way the Notifier detects push requests on the consolidated database.

See also: "server-initiated synchronization" on page 970.

PowerDesigner

A database modeling application. PowerDesigner provides a structured approach to designing a database or data warehouse. SQL Anywhere includes the Physical Data Model component of PowerDesigner.

PowerJ

A Sybase product for developing Java applications.

predicate

A conditional expression that is optionally combined with the logical operators AND and OR to make up the set of conditions in a WHERE or HAVING clause. In SQL, a predicate that evaluates to UNKNOWN is interpreted as FALSE.

primary key

A column or list of columns whose values uniquely identify every row in the table.

See also: "foreign key" on page 955.

primary key constraint

A uniqueness constraint on the primary key columns. A table can have only one primary key constraint.

See also:

- "constraint" on page 950
- "check constraint" on page 948
- "foreign key constraint" on page 956
- "unique constraint" on page 973
- "integrity" on page 958

primary table

The table containing the primary key in a foreign key relationship.

proxy table

A local table containing metadata used to access a table on a remote database server as if it were a local table.

See also: "metadata" on page 962.

publication

In MobiLink or SQL Remote, a database object that identifies data that is to be synchronized. In MobiLink, publications exist only on the clients. A publication consists of articles. SQL Remote users can receive a publication by subscribing to it. MobiLink users can synchronize a publication by creating a synchronization subscription to it.

See also:

- "replication" on page 968
- "article" on page 947
- "publication update" on page 966

publication update

In SQL Remote replication, a list of changes made to one or more publications in one database. A publication update is sent periodically as part of a replication message to the remote database(s).

See also:

- "replication" on page 968
- "publication" on page 965

publisher

In SQL Remote replication, the single user in a database who can exchange replication messages with other replicating databases.

See also: "replication" on page 968.

push notification

In QAnywhere, a special message delivered from the server to a QAnywhere client that prompts the client to initiate a message transmission.

See also: "QAnywhere" on page 966.

push request

In MobiLink server-initiated synchronization, a row in a SQL result set or table on the consolidated database that contains a notification and information about how to send the notification.

See also: "server-initiated synchronization" on page 970.

QAnywhere

Application-to-application messaging, including mobile device to mobile device and mobile device to and from the enterprise, that permits communication between custom programs running on mobile or wireless devices and a centrally located server application.

QAnywhere agent

In QAnywhere, a process running on the client device that monitors the client message store and determines when message transmission should occur.

query

A SQL statement or group of SQL statements that access and/or manipulate data in a database.

See also: "SQL" on page 970.

Redirector

A web server plug-in that routes requests and responses between a client and the MobiLink server. This plug-in also implements load-balancing and failover mechanisms.

reference database

In MobiLink, a SQL Anywhere database used in the development of UltraLite clients. You can use a single SQL Anywhere database as both reference and consolidated database during development. Databases made with other products cannot be used as reference databases.

referencing object

An object, such as a view, whose definition directly references another object in the database, such as a table.

See also: "foreign key" on page 955.

referenced object

An object, such as a table, that is directly referenced in the definition of another object, such as a view.

See also: "primary key" on page 965.

referential integrity

Adherence to rules governing data consistency, specifically the relationships between the primary and foreign key values in different tables. To have referential integrity, the values in each foreign key must correspond to the primary key values of a row in the referenced table.

See also:

- "primary key" on page 965
- "foreign key" on page 955

regular expression

A regular expression is a sequence of characters, wildcards, and operators that defines a pattern to search for within a string.

relational database management system (RDBMS)

A type of database management system that stores data in the form of related tables.

See also: "database management system (DBMS)" on page 952.

remote database

In MobiLink or SQL Remote, a database that exchanges data with a consolidated database. Remote databases may share all or some of the data in the consolidated database.

See also:

- "synchronization" on page 972
- "consolidated database" on page 950

remote DBA authority

In SQL Remote, a level of permission required by the Message Agent. In MobiLink, a level of permission required by the SQL Anywhere synchronization client (dbmlsync). When the Message Agent or synchronization client connects as a user who has this authority, it has full DBA access. The user ID has no additional permissions when not connected through the Message Agent or synchronization client.

See also: "DBA authority" on page 953.

remote ID

A unique identifier in SQL Anywhere and UltraLite databases that is used by MobiLink. The remote ID is initially set to NULL and is set to a GUID during a database's first synchronization.

replication

The sharing of data among physically distinct databases. Sybase has three replication technologies: MobiLink, SQL Remote, and Replication Server.

Replication Agent

See: "LTM" on page 960.

replication frequency

In SQL Remote replication, a setting for each remote user that determines how often the publisher's message agent should send replication messages to that remote user.

See also: "replication" on page 968.

replication message

In SQL Remote or Replication Server, a communication sent between a publishing database and a subscribing database. Messages contain data, passthrough statements, and information required by the replication system.

See also:

- "replication" on page 968
- "publication update" on page 966

Replication Server

A Sybase connection-based replication technology that works with SQL Anywhere and Adaptive Server Enterprise. It is intended for near-real time replication between a small number of databases.

See also: "LTM" on page 960.

role

In conceptual database modeling, a verb or phrase that describes a relationship from one point of view. You can describe each relationship with two roles. Examples of roles are "contains" and "is a member of."

role name

The name of a foreign key. This is called a role name because it names the relationship between the foreign table and primary table. By default, the role name is the table name, unless another foreign key is already using that name, in which case the default role name is the table name followed by a three-digit unique number. You can also create the role name yourself.

See also: "foreign key" on page 955.

rollback log

A record of the changes made during each uncommitted transaction. In the event of a ROLLBACK request or a system failure, uncommitted transactions are reversed out of the database, returning the database to its former state. Each transaction has a separate rollback log, which is deleted when the transaction is complete.

See also: "transaction" on page 972.

row-level trigger

A trigger that executes once for each row that is changed.

See also:

- "trigger" on page 973
- "statement-level trigger" on page 971

schema

The structure of a database, including tables, columns, and indexes, and the relationships between them.

script

In MobiLink, code written to handle MobiLink events. Scripts programmatically control data exchange to meet business needs.

See also: "event model" on page 955.

script-based upload

In MobiLink, a way to customize the upload process as an alternative to using the log file.

script version

In MobiLink, a set of synchronization scripts that are applied together to create a synchronization.

secured feature

A feature specified by the -sf option when a database server is started, so it is not available for any database running on that database server.

server-initiated synchronization

A way to initiate MobiLink synchronization programmatically from the consolidated database.

server management request

A QAnywhere message that is formatted as XML and sent to the QAnywhere system queue as a way to administer the sever message store or monitor QAnywhere applications.

server message store

In QAnywhere, a relational database on the server that temporarily stores messages until they are transmitted to a client message store or JMS system. Messages are exchanged between clients via the server message store.

service

In Windows operating systems, a way of running applications when the user ID running the application is not logged on.

session-based synchronization

A type of synchronization where synchronization results in consistent data representation across both the consolidated and remote databases. MobiLink is session-based.

snapshot isolation

A type of isolation level that returns a committed version of the data for transactions that issue read requests. SQL Anywhere provides three snapshot isolation levels: snapshot, statement-snapshot, and readonly-statement-snapshot. When using snapshot isolation, read operations do not block write operations.

See also: "isolation level" on page 958.

SQL

The language used to communicate with relational databases. ANSI has defined standards for SQL, the latest of which is SQL-2003. SQL stands, unofficially, for Structured Query Language.

SQL Anywhere

The relational database server component of SQL Anywhere that is intended for use in mobile and embedded environments or as a server for small and medium-sized businesses. SQL Anywhere is also the name of the package that contains the SQL Anywhere RDBMS, the UltraLite RDBMS, MobiLink synchronization software, and other components.

SQL-based synchronization

In MobiLink, a way to synchronize table data to MobiLink-supported consolidated databases using MobiLink events. For SQL-based synchronization, you can use SQL directly or you can return SQL using the MobiLink server APIs for Java and .NET.

SQL Remote

A message-based data replication technology for two-way replication between consolidated and remote databases. The consolidated and remote databases must be SQL Anywhere.

SQL statement

A string containing SQL keywords designed for passing instructions to a DBMS.

See also:

- "schema" on page 969
- "SQL" on page 970
- "database management system (DBMS)" on page 952

statement-level trigger

A trigger that executes after the entire triggering statement is completed.

See also:

- "trigger" on page 973
- "row-level trigger" on page 969

stored procedure

A program comprised of a sequence of SQL instructions, stored in the database and used to perform a particular task.

string literal

A string literal is a sequence of characters enclosed in single quotes.

subquery

A SELECT statement that is nested inside another SELECT, INSERT, UPDATE, or DELETE statement, or another subquery.

There are two types of subquery: correlated and nested.

subscription

In MobiLink synchronization, a link in a client database between a publication and a MobiLink user, allowing the data described by the publication to be synchronized.

In SQL Remote replication, a link between a publication and a remote user, allowing the user to exchange updates on that publication with the consolidated database.

See also:

- "publication" on page 965
- "MobiLink user" on page 962

Sybase Central

A database management tool that provides SQL Anywhere database settings, properties, and utilities in a graphical user interface. Sybase Central can also be used for managing other Sybase products, including MobiLink.

synchronization

The process of replicating data between databases using MobiLink technology.

In SQL Remote, synchronization is used exclusively to denote the process of initializing a remote database with an initial set of data.

See also:

- "MobiLink" on page 962
- "SQL Remote" on page 971

SYS

A special user that owns most of the system objects. You cannot log in as SYS.

system object

Database objects owned by SYS or dbo.

system table

A table, owned by SYS or dbo, that holds metadata. System tables, also known as data dictionary tables, are created and maintained by the database server.

system view

A type of view, included in every database, that presents the information held in the system tables in an easily understood format.

temporary table

A table that is created for the temporary storage of data. There are two types: global and local.

See also:

- "local temporary table" on page 960
- "global temporary table" on page 957

transaction

A sequence of SQL statements that comprise a logical unit of work. A transaction is processed in its entirety or not at all. SQL Anywhere supports transaction processing, with locking features built in to allow concurrent transactions to access the database without corrupting the data. Transactions end either with a COMMIT statement, which makes the changes to the data permanent, or a ROLLBACK statement, which undoes all the changes made during the transaction.

transaction log

A file storing all changes made to a database, in the order in which they are made. It improves performance and allows data recovery in the event the database file is damaged.

transaction log mirror

An optional identical copy of the transaction log file, maintained simultaneously. Every time a database change is written to the transaction log file, it is also written to the transaction log mirror file.

A mirror file should be kept on a separate device from the transaction log, so that if either device fails, the other copy of the log keeps the data safe for recovery.

See also: "transaction log" on page 973.

transactional integrity

In MobiLink, the guaranteed maintenance of transactions across the synchronization system. Either a complete transaction is synchronized, or no part of the transaction is synchronized.

transmission rule

In QAnywhere, logic that determines when message transmission is to occur, which messages to transmit, and when messages should be deleted.

trigger

A special form of stored procedure that is executed automatically when a user runs a query that modifies the data.

See also:

- "row-level trigger" on page 969
- "statement-level trigger" on page 971
- "integrity" on page 958

UltraLite

A database optimized for small, mobile, and embedded devices. Intended platforms include cell phones, pagers, and personal organizers.

UltraLite runtime

An in-process relational database management system that includes a built-in MobiLink synchronization client. The UltraLite runtime is included in the libraries used by each of the UltraLite programming interfaces, as well as in the UltraLite engine.

unique constraint

A restriction on a column or set of columns requiring that all non-null values are different. A table can have multiple unique constraints.

See also:

- "foreign key constraint" on page 956
- "primary key constraint" on page 965
- "constraint" on page 950

unload

Unloading a database exports the structure and/or data of the database to text files (SQL command files for the structure, and ASCII comma-separated files for the data). You unload a database with the Unload utility.

In addition, you can unload selected portions of your data using the UNLOAD statement.

upload

The stage in synchronization where data is transferred from a remote database to a consolidated database.

user-defined data type

See "domain" on page 954.

validate

To test for particular types of file corruption of a database, table, or index.

view

A SELECT statement that is stored in the database as an object. It allows users to see a subset of rows or columns from one or more tables. Each time a user uses a view of a particular table, or combination of tables, it is recomputed from the information stored in those tables. Views are useful for security purposes, and to tailor the appearance of database information to make data access straightforward.

window

The group of rows over which an analytic function is performed. A window may contain one, many, or all rows of data that has been partitioned according to the grouping specifications provided in the window definition. The window moves to include the number or range of rows needed to perform the calculations for the current row in the input. The main benefit of the window construct is that it allows additional opportunities for grouping and analysis of results, without having to perform additional queries.

Windows

The Microsoft Windows family of operating systems, such as Windows Vista, Windows XP, and Windows 200x.

Windows CE

See "Windows Mobile" on page 974.

Windows Mobile

A family of operating systems produced by Microsoft for mobile devices.

work table

An internal storage area for interim results during query optimization.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X