Connecting to Your Database

Appeon PowerBuilder®

2017

DOCUMENT ID: DC37776-01-1700-01

LAST REVISED: July 2017

Contents

About This Book

Audience

This book is for anyone who uses PowerBuilder® to connect to a database. It assumes that you are familiar with the database you are using and have installed the server and client software required to access the data.

How to use this book

This book describes how to connect to a database in PowerBuilder by using a standard or native database interface. It gives procedures for preparing, defining, establishing, maintaining, and troubleshooting your database connections. For an overview of the steps you need to take, see Basic connection procedure on page 3.

Related documents

For detailed information about supported database interfaces, DBParm parameters, and database preferences, see the Database Connectivity section in the PowerBuilder Help. For a complete list of PowerBuilder documentation, see PowerBuilder *Getting Started*.

Other sources of information

Use the Appeon Product Manuals web site to learn more about your product. The Appeon Product Manuals web site is accessible using a standard Web browser.

To access the Appeon Product Manuals web site, go to Product Manuals at https://www.appeon.com/developers/library/product-manuals-for-pb.

Conventions

The formatting conventions used in this manual are:

Formatting example	Indicates
Retrieve and Update	When used in descriptive text, this font indicates: • Command, function, and method names • Keywords such as true, false, and null • Datatypes such as integer and char • Database column names such as emp_id and f_name • User-defined objects such as dw_emp or w_main

Formatting example	Indicates
variable or *file name*	When used in descriptive text and syntax descriptions, oblique font indicates: • Variables, such as *myCounter* • Parts of input text that must be substituted, such as *pblname*`.pbd` • File and path names
File>Save	Menu names and menu items are displayed in plain text. The greater than symbol (>) shows you how to navigate menu selections. For example, File>Save indicates "select Save from the File menu."
`dw_1.Update()`	Monospace font indicates: • Information that you enter in a dialog box or on a command line • Sample script fragments • Sample output fragments

If you need help

All customers are entitled to standard technical support for reproducible software defects. You can open a standard support ticket at the Appeon support site: https://www.appeon.com/standardsupport/ (login required).

If your organization has purchased a premium support contract for this product, then the designated authorized support contact(s) may seek assistance with your technical issue or question at the Appeon support site: https://support.appeon.com (login required).

PART 1

Introduction to Database Connections

This part introduces data connections in PowerBuilder. It helps you understand how to connect to a database in the PowerBuilder development environment.

CHAPTER 1 Understanding Data Connections

About this chapter

This chapter gives an overview of the concepts and procedures for connecting to a database in the PowerBuilder development environment.

Contents

How to find the information you need

When you work with PowerBuilder, you can connect to a database in the development environment or in an application script.

This book describes how to connect to your database in the PowerBuilder development environment.

For information about connecting to a database in a PowerBuilder application script, see *Application Techniques*.

Basic connection procedure

The following table gives an overview of the connection procedure and indicates where you can find detailed information about each step.

Table 1-1: Basic connection procedure

Step	Action	Details	See
1	(Optional) Get an introduction to database connections in PowerBuilder	If necessary, learn more about how PowerBuilder connects to a database in the development environment	Chapter 1 (this chapter)

Step	Action	Details	See
2	Prepare to use the data source or database before connecting to it for the first time in PowerBuilder	Outside PowerBuilder, install the required network, database server, and database client software and verify that you can connect to the database	*For ODBC data sources*: Chapter 2, Using the ODBC Interface *For JDBC data sources*: Chapter 3, Using the JDBC Interface *For OLE DB data sources*: Chapter 4, Using the OLE DB Interface *For ADO.NET data sources*: Chapter 5, Using the ADO.NET Interface *For native database interfaces*: Chapter 7, Using Native Database Interfaces
3	Install the ODBC driver, OLE DB data provider, ADO.NET data provider, or native database interface	Install the driver, database provider, or native database interface required to access your data	For a list of what is supported on your platform: "Supported Database Interfaces" in online Help
4	Define the data source (ODBC connections and some OLE DB drivers)	Create the required configuration for a data source accessed through ODBC	*For ODBC data sources*: Chapter 2, Using the ODBC Interface
5	Define the database interface	Create the database profile	*For ODBC data sources*: Chapter 2, Using the ODBC Interface *For JDBC data sources*: Chapter 3, Using the JDBC Interface *For OLE DB data sources*: Chapter 4, Using the OLE DB Interface *For ADO.NET data sources*: Chapter 5, Using the ADO.NET Interface *For native database interfaces*: Chapter 7, Using Native Database Interfaces
7	Connect to the data source or database	Access the data in PowerBuilder	Chapter 13, Managing Database Connections
8	(Optional) Set additional connection parameters	If necessary, set DBParm parameters and database preferences to fine-tune your database connection and take advantage of DBMS-specific features that your interface supports	*For procedures*: Chapter 14, Setting Additional Connection Parameters *For DBParm descriptions*: online Help *For database preference descriptions*: online Help

Step	Action	Details	See
9	(Optional) Troubleshoot the data connection	If necessary, use the trace tools to troubleshoot problems with your connection	Chapter 15, Troubleshooting Your Connection

Accessing data in PowerBuilder

There are several ways to access data in the PowerBuilder development environment:

- Through one of the standard database interfaces such as ODBC, JDBC,ADO.NET, or OLE DB
- Through one of the native database interfaces

Standard database interfaces

A standard database interface communicates with a database through a standard-compliant driver (in the case of ODBC and JDBC) or data provider (in the case of OLE DB and ADO.NET). The standard-compliant driver or data provider translates the abstract function calls defined by the standard's API into calls that are understood by a specific database. To use a standard interface, you need to install the standard's API and a suitable driver or data provider. Then, install the standard database interface you want to use to access your DBMS by selecting the interface in the PowerBuilder Setup program.

PowerBuilder currently supports the following standard interfaces:

- Open Database Connectivity (ODBC)
- Java Database Connectivity (JDBC)
- Microsoft's Universal Data Access Component OLE DB
- Microsoft's ADO.NET

Native database interfaces

A native database interface communicates with a database through a direct connection. It communicates to a database using that database's native API.

To access data through one of the native database interfaces, you must first install the appropriate database software on the server and client workstations at your site. Then, install the native database interface that accesses your DBMS by selecting the interface in the PowerBuilder Setup program.

For example, if you have the appropriate SAP Adaptive Server® Enterprise server and client software installed, you can access the database by installing the Adaptive Server Enterprise database interface.

Loading database interface libraries

PowerBuilder loads the libraries used by a database interface when it connects to the database. PowerBuilder does *not* automatically free the database interface libraries when it disconnects.

Although memory use is somewhat increased by this technique (since the loaded database interface libraries continue to be held in memory), the technique improves performance and eliminates problems associated with the freeing and subsequent reloading of libraries experienced by some database connections.

If you want PowerBuilder to free database interface libraries on disconnecting from the database (as it did prior to PowerBuilder 8), you can change its default behavior:

To change the default behavior for	Do this
Connections in the development environment	Select the Free Database Driver Libraries On Disconnect check box on the General tab of the System Options dialog box
Runtime connections	Set the FreeDBLibraries property of the Application object to TRUE on the General tab of the Properties view in the Application painter or in a script

Accessing the Demo Database

PowerBuilder includes a standalone SQL Anywhere® database called the Demo Database. Unless you clear this option in the setup program, the database is installed automatically. You access tables in the Demo Database when you use the PowerBuilder tutorial.

A SQL Anywhere database is considered an ODBC data source, because you access it with the SQL Anywhere ODBC driver.

Using database profiles

What is a database profile?

A database profile is a named set of parameters stored in your system registry that defines a connection to a particular database in the PowerBuilder development environment. You must create a database profile for each data connection.

What you can do

Using database profiles is the easiest way to manage data connections in the PowerBuilder development environment. For example, you can:

- Select a database profile to connect to or switch between databases
- Edit a database profile to customize a connection
- Delete a database profile if you no longer need to access that data
- Import and export database profiles to share connection parameters quickly

For more information

For instructions on using database profiles, see Chapter 13, Managing Database Connections.

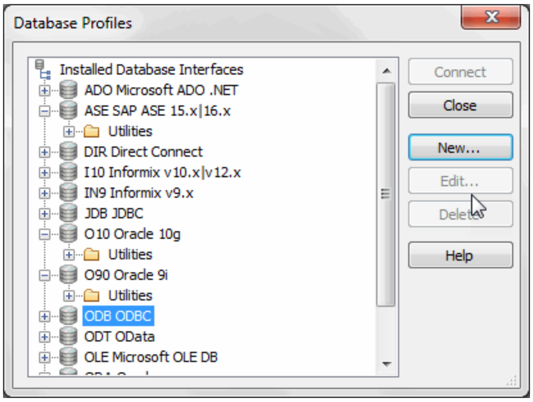
About creating database profiles

You work with two dialog boxes when you create a database profile in PowerBuilder: the Database Profiles dialog box and the interface-specific Database Profile Setup dialog box.

Using the Database painter to create database profiles
You can also create database profiles from the Database painter's Objects view.

Database Profiles dialog box

The Database Profiles dialog box uses an easy-to-navigate tree control format to display your installed database interfaces and defined database profiles. You can create, edit, and delete database profiles from this dialog box.



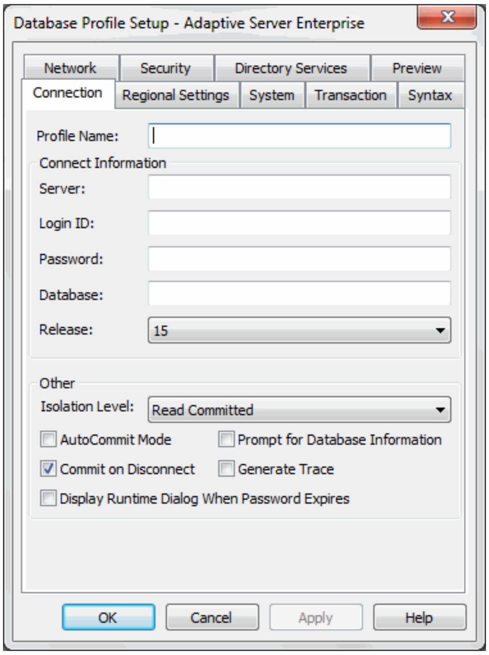
When you run the PowerBuilder Setup program, it updates the Vendors list in the PowerBuilder® section in the HKEY_LOCAL_MACHINE registry key with the interfaces you install. The Database Profiles dialog box displays the same interfaces that appear in the Vendors list.

Where the Vendors list is stored
The *Sybase\PowerBuilder\17.0\Vendors* key in *HKEY_LOCAL_MACHINE\SOFTWARE* is used for InfoMaker as well as PowerBuilder.

For detailed instructions on using the Database Profiles dialog box to connect to a database and manage your profiles, see Chapter 13, Managing Database Connections.

Database Profile Setup dialog box

Each database interface has its own Database Profile Setup dialog box where you can set interface-specific connection parameters. For example, if you install the Adaptive Server Enterprise ASE interface and then select it and click New in the Database Profiles dialog box, the Database Profile Setup - Adaptive Server Enterprise dialog box displays, containing settings for the connection options that apply to this interface.


Database Profile Setup - Adaptive Server Enterprise

Profile Name:
Connect Information
Server:
Login ID:
Password:
Database:
Release: 15
Other
Isolation Level: Read Committed
☐ AutoCommit Mode
☐ Prompt for Database Information
☑ Commit on Disconnect
☐ Generate Trace
☐ Display Runtime Dialog When Password Expires

The Database Profile Setup dialog box groups similar connection parameters on the same tab page and lets you easily set their values by using check boxes, drop-down lists, and text boxes. Basic (required) connection parameters are on the Connection tab page, and additional connection options (DBParm parameters and SQLCA properties) are on the other tab pages.

As you complete the Database Profile Setup dialog box in PowerBuilder, the correct PowerScript® connection syntax for each selected option is generated on the Preview tab. You can copy the syntax you want from the Preview tab into a PowerBuilder application script.

Supplying sufficient information in the Database Profile Setup dialog box

For some database interfaces, you might not need to supply values for all boxes in the Database Profile Setup dialog box. If you supply the profile name and click OK, PowerBuilder displays a series of dialog boxes to prompt you for additional information when you connect to the database.

This information can include:

- User ID or login ID
- Password or login password
- Database name
- Server name

For some databases, supplying only the profile name does not give PowerBuilder enough information to prompt you for additional connection values. For these interfaces, you must supply values for all applicable boxes in the Database Profile Setup dialog box.

For information about the values you should supply for your connection, click Help in the Database Profile Setup dialog box for your interface.

Creating a database profile

To create a new database profile for a database interface, you must complete the Database Profile Setup dialog box for the interface you are using to access the database.

❖ **To create a database profile for a database interface:**

1. Click the Database Profile button in the PowerBar.

   The Database Profiles dialog box displays, listing your installed database interfaces. To see a list of database profiles defined for a particular interface, click the plus sign to the left of the interface name or double-click the interface name to expand the list.

2 Highlight an interface name and click New.

The Database Profile Setup dialog box for the selected interface displays. For example, if you select the SYC interface, the Database Profile Setup - Adaptive Server Enterprise dialog box displays.

Client software and interface must be installed
To display the Database Profile Setup dialog box for your interface, the required client software and native database interface must be properly installed and configured. For specific instructions for your database interface, see the chapter on using the interface.

3 On the Connection tab page, type the profile name and supply values for any other basic parameters your interface requires to connect.

For information about the basic connection parameters for your interface and the values you should supply, click Help.

About the DBMS identifier
You do *not* need to specify the DBMS identifier in a database profile. When you create a new profile for any installed database interface, PowerBuilder generates the correct DBMS connection syntax for you.

4 (Optional) On the other tab pages, supply values for any additional connection options (DBParm parameters and SQLCA properties) to take advantage of DBMS-specific features that your interface supports.

For information about the additional connection parameters for your interface and the values you should supply, click Help.

5 (Optional) Click the Preview tab if you want to see the PowerScript connection syntax that PowerBuilder generates for each selected option.

You can copy the PowerScript connection syntax from the Preview tab directly into a PowerBuilder application script.

For instructions on using the Preview tab to help you connect in a PowerBuilder application, see the section on using Transaction objects in *Application Techniques*.

6 Click OK to save your changes and close the Database Profile Setup dialog box. (To save your changes on a particular tab page *without* closing the dialog box, click Apply.)

The Database Profiles dialog box displays, with the new profile name highlighted under the appropriate interface. The database profile values are saved in the system registry in *HKEY_CURRENT_USER\Software\Sybase\PowerBuilder\17.0\Database Profiles\PowerBuilder*.

You can look at the registry entry or export the profile as described in Importing and exporting database profiles on page 177 to see the settings you made. The NewLogic parameter is set to True by default. This setting specifies that the password is encrypted using Unicode encoding.

What to do next

For instructions on preparing to use and then defining an ODBC data source, see Chapter 2, Using the ODBC Interface.

For instructions on preparing to use and then defining a JDBC database interface, see Chapter 3, Using the JDBC Interface.

For instructions on preparing to use and then defining an OLE DB data provider, see Chapter 4, Using the OLE DB Interface.

For instructions on preparing to use and then defining an ADO.NET data provider, see Chapter 5, Using the ADO.NET Interface.

For instructions on preparing to use and then defining a native database interface, see Chapter 7, Using Native Database Interfaces.

PART 2

Working with Standard Database Interfaces

This part describes how to set up and define database connections accessed through one of the standard database interfaces.

CHAPTER 2 Using the ODBC Interface

About this chapter

This chapter gives an introduction to the ODBC interface and then describes how to prepare to use the data source, how to define the data source, and how to define the ODBC database profile. It also describes how to use the SAP SQL Anywhere ODBC driver.

Contents

For more information

This chapter gives general information about preparing to use and defining each ODBC data source. For more detailed information, use the online Help provided by the driver vendor, as described in Displaying Help for ODBC drivers on page 28. This Help provides important details about using the data source.

About the ODBC interface

You can access a wide variety of ODBC data sources in PowerBuilder. This section describes what you need to know to use ODBC connections to access your data in PowerBuilder.

What is ODBC?

The ODBC API

Open Database Connectivity (**ODBC**) is a standard application programming interface (API) developed by Microsoft. It allows a single application to access a variety of data sources for which ODBC-compliant drivers exist. The application uses Structured Query Language (SQL) as the standard data access language.

The ODBC API defines the following:

- A library of ODBC function calls that connect to the data source, execute SQL statements, and retrieve results
- A standard way to connect and log in to a DBMS
- SQL syntax based on the X/Open and SQL Access Group (SAG) CAE specification (1992)
- A standard representation for datatypes
- A standard set of error codes

Accessing ODBC data sources

Applications that provide an ODBC interface, like PowerBuilder, can access data sources for which an ODBC driver exists. An **ODBC data source driver** is a dynamic link library (DLL) that implements ODBC function calls. The application invokes the ODBC driver to access a particular data source.

Accessing Unicode data

Using the ODBC interface, PowerBuilder can connect, save, and retrieve data in both ANSI/DBCS and Unicode databases but does not convert data between Unicode and ANSI/DBCS. When character data or command text is sent to the database, PowerBuilder sends a Unicode string. The driver must guarantee that the data is saved as Unicode data correctly. When PowerBuilder retrieves character data, it assumes the data is Unicode.

A Unicode database is a database whose character set is set to a Unicode format, such as UTF-8, UTF-16, UCS-2, or UCS-4. All data must be in Unicode format, and any data saved to the database must be converted to Unicode data implicitly or explicitly.

A database that uses ANSI (or DBCS) as its character set might use special datatypes to store Unicode data. Columns with these datatypes can store *only* Unicode data. Any data saved into such a column must be converted to Unicode explicitly. This conversion must be handled by the database server or client.

Using ODBC in PowerBuilder

What you can do

The following ODBC connectivity features are available in PowerBuilder:

- Connect to a SQL Anywhere standalone database (including the Demo Database) using the SQL Anywhere ODBC driver and the ODBC interface
- Create and delete local SQL Anywhere databases

For instructions, see the *Users Guide*.

- Connect to an installed SAP IQ database client through the ODBC interface.
- Use Level 1 or later ODBC-compliant drivers obtained from vendors other than SAP to access your data

  See Obtaining ODBC drivers on page 23.
- Use Microsoft's ODBC Data Source Administrator to define ODBC data sources

  See Defining ODBC data sources on page 25.

Components of an ODBC connection

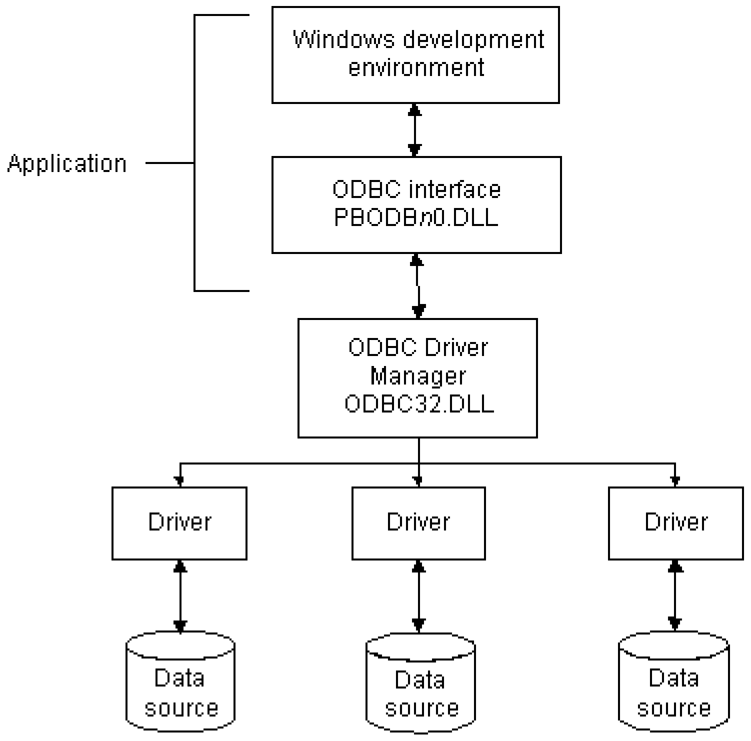
How an ODBC connection is made

When you access an ODBC data source in PowerBuilder, your connection goes through several layers before reaching the data source. It is important to understand that each layer represents a separate component of the connection, and that each component might come from a different vendor.

Because ODBC is a standard API, PowerBuilder uses the same interface to access every ODBC data source. As long as a driver is ODBC compliant, PowerBuilder can access it through the ODBC interface to the ODBC Driver Manager. The development environment and the ODBC interface work together as the application component.

Figure 2-1 shows the general components of an ODBC connection.

Figure 2-1: Components of an ODBC connection


Application
Windows development environment
ODBC interface PBODB*n*0.DLL
ODBC Driver Manager ODBC32.DLL
Driver
Driver
Driver
Data source
Data source
Data source

Component descriptions

Table 2-1 gives the provider and a brief description of each ODBC component shown in the diagram.

Table 2-1: Provider and function of ODBC connection components

Component	Provider	What it does
Application	SAP	Calls ODBC functions to submit SQL statements, catalog requests, and retrieve results from a data source. PowerBuilder uses the same ODBC interface to access all ODBC data sources.
ODBC Driver Manager	Microsoft	Installs, loads, and unloads drivers for an application.

Component	Provider	What it does
Driver	Driver vendor	Processes ODBC function calls, submits SQL requests to a particular data source, and returns results to an application. If necessary, translates an application's request so that it conforms to the SQL syntax supported by the back-end database. See Types of ODBC drivers next.
Data source	DBMS or database vendor	Stores and manages data for an application. Consists of the data to be accessed and its associated DBMS, operating system, and (if present) network software that accesses the DBMS.

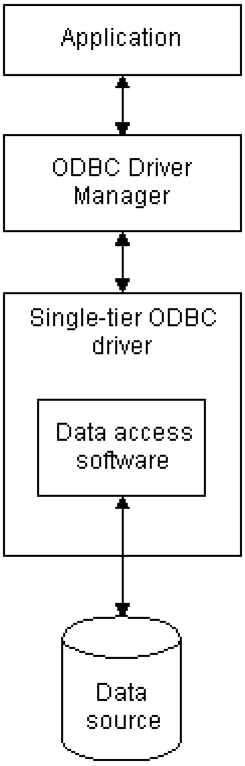
Types of ODBC drivers

When PowerBuilder is connected to an ODBC data source, you might see messages from the ODBC driver that include the words *single-tier* or *multiple-tier*. These terms refer to the two types of drivers defined by the ODBC standard.

Single-tier driver

A single-tier ODBC driver processes both ODBC functions and SQL statements. In other words, a single-tier driver includes the data access software required to manage the data source file and catalog tables. An example of a single-tier ODBC driver is the Microsoft Access driver.

Figure 2-2: Single-tier ODBC driver

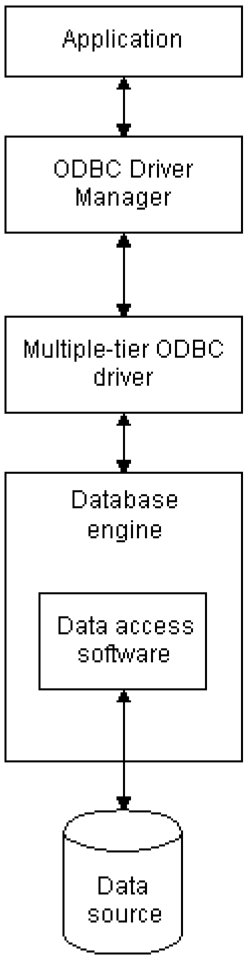


Multiple-tier driver

A multiple-tier ODBC driver processes ODBC functions, but sends SQL statements to the database engine for processing. Unlike the single-tier driver, a multiple-tier driver does not include the data access software required to manage the data directly.

An example of a multiple-tier ODBC driver is the SAP SQL Anywhere driver.

Figure 2-3: Multi-tier ODBC driver



Ensuring the proper ODBC driver conformance levels

You can access data in PowerBuilder with ODBC drivers obtained from vendors other than SAP, such as DBMS vendors.

An ODBC driver obtained from another vendor must meet certain conformance requirements to ensure that it works properly with PowerBuilder. This section describes how to make sure your driver meets these requirements.

What are ODBC conformance levels?

PowerBuilder can access many data sources for which ODBC-compliant drivers exist. However, ODBC drivers manufactured by different vendors might vary widely in the functions they provide.

To ensure a standard level of compliance with the ODBC interface, and to provide a means by which application vendors can determine whether a specific driver provides the functions they need, ODBC defines conformance levels for drivers in two areas:

- **API** Deals with supported ODBC function calls
- **SQL grammar** Deals with supported SQL statements and SQL datatypes

API conformance levels

ODBC defines three API conformance levels, in order of increasing functionality:

- **Core** A set of core API functions that corresponds to the functions in the ISO Call Level Interface (CLI) and X/Open CLI specification
- **Level 1** Includes all Core API functions and several extended functions usually available in an OLTP relational DBMS
- **Level 2** Includes all Core and Level 1 API functions and additional extended functions

❖ **To ensure the proper ODBC driver API conformance level:**

- Appeon recommends that the ODBC drivers you use with PowerBuilder meet *Level 1 or higher* API conformance requirements. However, PowerBuilder might also work with drivers that meet Core level API conformance requirements.

SQL conformance levels

ODBC defines three SQL grammar conformance levels, in order of increasing functionality:

- **Minimum** A set of SQL statements and datatypes that meets a basic level of ODBC conformance
- **Core** Includes all Minimum SQL grammar and additional statements and datatypes that roughly correspond to the X/Open and SAG CAE specification (1992)
- **Extended** Includes all Minimum and Core SQL grammar and an extended set of statements and datatypes that support common DBMS extensions to SQL

❖ **To ensure the proper ODBC driver SQL conformance level:**

- Appeon recommends that the ODBC drivers you use with PowerBuilder meet *Core or higher* SQL conformance requirements. However, PowerBuilder might also work with drivers that meet Minimum level SQL conformance requirements.

Obtaining ODBC drivers

You can use the ODBC driver for the SQL Anywhere® developer edition from SAP, provided with PowerBuilder, to access data. Other SAP database clients also include ODBC drivers that you can access through the PowerBuilder ODBC interface. See your database documentation for details.

PowerBuilder also let you access data with *any* Level 1 or higher ODBC-compliant drivers obtained from a vendor other than SAP. In most cases, these drivers will work with PowerBuilder.

Using ODBC drivers with PowerBuilder

Using existing Microsoft ODBC drivers

If you already have version 2.0 or later of any of the following Microsoft ODBC drivers installed and properly configured, you can use these drivers with PowerBuilder to connect to your data source:

Microsoft Access (*.MDB)
Microsoft Btrieve (*.DDF)
Microsoft dBASE (*.DBF)
Microsoft Excel (*.XLS)
Microsoft FoxPro (*.DBF)
Microsoft Paradox (*.DB)
Microsoft Text (*.CSV, *.TXT)

Getting help with ODBC drivers

To ensure that you have up-to-date and accurate information about using your ODBC driver with PowerBuilder, get help as needed by doing one or more of the following:

To get help on	Do this
Using the ODBC Data Source Administrator	Click the Help button on each tab.

To get help on	Do this
Completing the ODBC setup dialog box for your driver	Click the Help button (if present) in the ODBC setup dialog box for your driver.
Using SQL Anywhere	See the SQL Anywhere documentation.
Using an ODBC driver obtained from a vendor other than SAP	See the vendor's documentation for that driver.
Troubleshooting your ODBC connection	Check for a technical document that describes how to connect to your ODBC data source.

Preparing ODBC data sources

The first step in connecting to an ODBC data source is preparing the data source. This ensures that you are able to connect to the data source and use your data in PowerBuilder.

You prepare to use a data source *outside* PowerBuilder *before* you start the product, define the data source, and connect to it. The requirements differ for each data source, but in general, preparing to use a data source involves the following steps.

❖ To prepare to use an ODBC data source with PowerBuilder:

1. If network software is required to access the data source, make sure it is properly installed and configured at your site and on the client workstation.

2. If database software is required, make sure it is properly installed and configured on your computer or network server.

3. Make sure the required data files are present on your computer or network server.

4. Make sure the names of tables and columns you want to access follow standard SQL naming conventions.

   Avoid using blank spaces or database-specific reserved words in table and column names. Be aware of the case-sensitivity options of the DBMS. It is safest to use all uppercase characters when naming tables and columns that you want to access in PowerBuilder.

5. If your database requires it, make sure the tables you want to access have unique indexes.

6 Install both of the following using the PowerBuilder Setup program:

- The ODBC driver that accesses your data source
- The ODBC interface

Defining ODBC data sources

Each ODBC data source requires a corresponding ODBC driver to access it. When you define an ODBC data source, you provide information about the data source that the driver requires in order to connect to it. Defining an ODBC data source is often called configuring the data source.

After you prepare to use the data source, you must define it using Microsoft's ODBC Data Source Administrator utility. This utility can be accessed from the Control Panel in Windows or PowerBuilder's Database painter.

The rest of this section describes what you need to know to define an ODBC data source in order to access it in the PowerBuilder development environment.

How PowerBuilder accesses the data source

When you access an ODBC data source in PowerBuilder, there are several initialization files and registry entries on your computer that work with the ODBC interface and driver to make the connection.

PBODB170 initialization file

Contents

PBODB170.INI is installed in the *Appeon\Shared\PowerBuilder* directory. The first time the user opens PowerBuilder, the file is copied to *AppData\Local\Appeon*\PowerBuilder *17.0* in the user's profile folder (for example, under *C:\users\<username>*). This copy is used when running PowerBuilder.PowerBuilder uses *PBODB170.INI* to maintain access to extended functionality in the back-end DBMS, for which ODBC does not provide an API call. Examples of extended functionality are SQL syntax or DBMS-specific function calls.

Editing

In most cases, you do not need to edit *PBODB170.INI*. In certain situations, however, you might need to add functions to *PBODB170.INI* for your back-end DBMS. Be sure to edit the copy in your user profile folder, not the original copy.

For instructions, see the Appendix, Adding Functions to the PBODB170 Initialization File.

ODBCINST registry entries

Contents

The ODBCINST initialization information is located in the *HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI* registry key. When you install an ODBC-compliant driver, *ODBCINST.INI* is automatically updated with a description of the driver.

This description includes:

- The DBMS or data source associated with the driver
- The drive and directory of the driver and setup DLLs (for some data sources, the driver and setup DLLs are the same)
- Other driver-specific connection parameters

Editing

You do *not* need to edit the registry key directly to modify connection information. If your driver uses the information in the *ODBCINST.INI* registry key, the key is automatically updated when you install the driver. This is true whether the driver is supplied by SAP or another vendor.

ODBC registry entries

Contents

ODBC initialization information is located in the *HKEY_CURRENT_USER\SOFTWARE\ODBC\ODBC.INI* registry key. When you define a data source for a particular ODBC driver, the driver writes the values you specify in the ODBC setup dialog box to the *ODBC.INI* registry key.

The *ODBC.INI* key contains subkeys named for each defined data source. Each subkey contains the values specified for that data source in the ODBC setup dialog box. The values might vary for each data source but generally include the following:

- Database
- Driver

- Optional description
- DBMS-specific connection parameters

Editing Do *not* edit the *ODBC* subkey directly to modify connection information. Instead, use a tool designed to define ODBC data sources and the ODBC configuration automatically, such as the ODBC Data Source Administrator.

Database profiles registry entry

Contents Database profiles for all data sources are stored in the registry in *HKEY_CURRENT_USER\SOFTWARE\Sybase\PowerBuilder\17.0\Database Profiles*.

Editing You should *not* need to edit the profiles directly to modify connection information. These files are updated automatically when PowerBuilder creates the database profile as part of the ODBC data source definition.

You can also edit the profile in the Database Profile Setup dialog box or complete the Database Preferences dialog box in PowerBuilder to specify other connection parameters stored in the registry. (For instructions, see Chapter 14, Setting Additional Connection Parameters.)

Example The following example shows a portion of the database profile for an Demo Database data source:

```
DBMS=ODBC
DBParm=ConnectString='DSN=PB Demo DB V2017
DB;UID=dba;PWD=00c61737'
Prompt=0
```

This registry entry example shows the two most important values in a database profile for an ODBC data source:

- **DBMS** The DBMS value (ODBC) indicates that you are using the ODBC interface to connect to the data source.
- **DBParm** The ConnectString DBParm parameter controls your ODBC data source connection. The connect string *must* specify the DSN (data source name) value, which tells ODBC which data source you want to connect to. When you select a database profile to connect to a data source, ODBC looks in the ODBC.INI registry key for a subkey that corresponds to the data source name in your profile. ODBC then uses the information in the subkey to load the required libraries to connect to the data source. The connect string can also contain the UID (user ID) and PWD (password) values needed to access the data source.

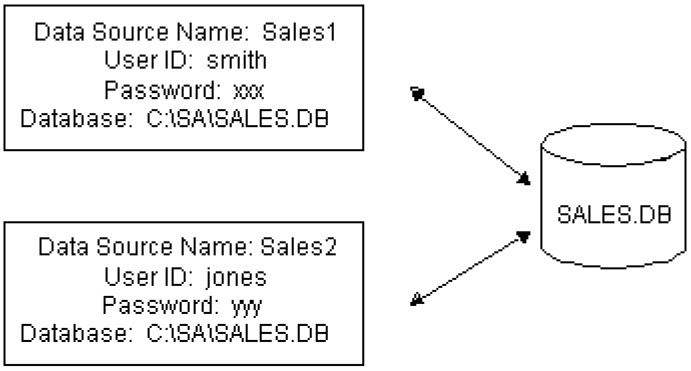
Defining multiple data sources for the same data

When you define an ODBC data source in PowerBuilder, each data source name must be unique. You can, however, define multiple data sources that access the same data, as long as the data sources have unique names.

For example, assume that your data source is a SQL Anywhere database located in *C:\SQL Anywhere\SALES.DB*. Depending on your application, you might want to specify different sets of connection parameters for accessing the database, such as different passwords and user IDs.

To do this, you can define two ODBC data sources named Sales1 and Sales2 that specify the same database (*C:\SQL Anywhere\SALES.DB*) but use different user IDs and passwords. When you connect to the data source using a profile created for either of these data sources, you are using different connection parameters to access the same data.

Figure 2-4: Using two data sources to access a database



Displaying Help for ODBC drivers

The online Help for ODBC drivers in PowerBuilder is provided by the driver vendors. It gives help on:

- Completing the ODBC setup dialog box to define the data source
- Using the ODBC driver to access the data source

Help for any ODBC driver

Use the following procedure to display vendor-supplied Help when you are in the ODBC setup dialog box for ODBC drivers.

❖ **To display Help for any ODBC driver:**

- Click the Help button in the ODBC setup dialog box for your driver.

  A Help window displays, describing features in the setup dialog box.

Selecting an ODBC translator

What is an ODBC translator?

Some ODBC drivers allow you to specify a translator when you define the data source. An **ODBC translator** is a DLL that translates data passing between an application and a data source. Typically, translators are used to translate data from one character set to another.

What you do

Follow these steps to select a translator for your ODBC driver.

❖ **To select a translator when using an ODBC driver:**

1 In the ODBC setup dialog box for your driver, display the Select Translator dialog box.

  The way you display the Select Translator dialog box depends on the driver and Windows platform you are using. Click Help in your driver's setup dialog box for instructions on displaying the Select Translator dialog box.

  In the Select Translator dialog box, the translators listed are determined by the values in your *ODBCINST.INI* registry key.

2 From the Installed Translators list, select a translator to use.

  If you need help using the Select Translator dialog box, click Help.

3 Click OK.

  The Select Translator dialog box closes and the driver performs the translation.

Defining the ODBC interface

To define a connection through the ODBC interface, you must create a database profile by supplying values for at least the basic connection parameters in the Database Profile Setup - ODBC dialog box. You can then select this profile at any time to connect to your data source in the development environment.

For information on how to define a database profile, see Using database profiles on page 6.

SAP SQL Anywhere

This section describes how to prepare and define an SAP SQL Anywhere data source in order to connect to it using the SQL Anywhere ODBC driver.

Name change
For versions 6 through 9, the SQL Anywhere database server was called Adaptive Server® Anywhere (ASA). However in this documentation, we call all of them SQL Anywhere for the sake of consistence.

SQL Anywhere includes two database servers—a personal database server and a network database server. For information about using SAP SQL Anywhere, see the SQL Anywhere documentation.

Supported versions for SQL Anywhere

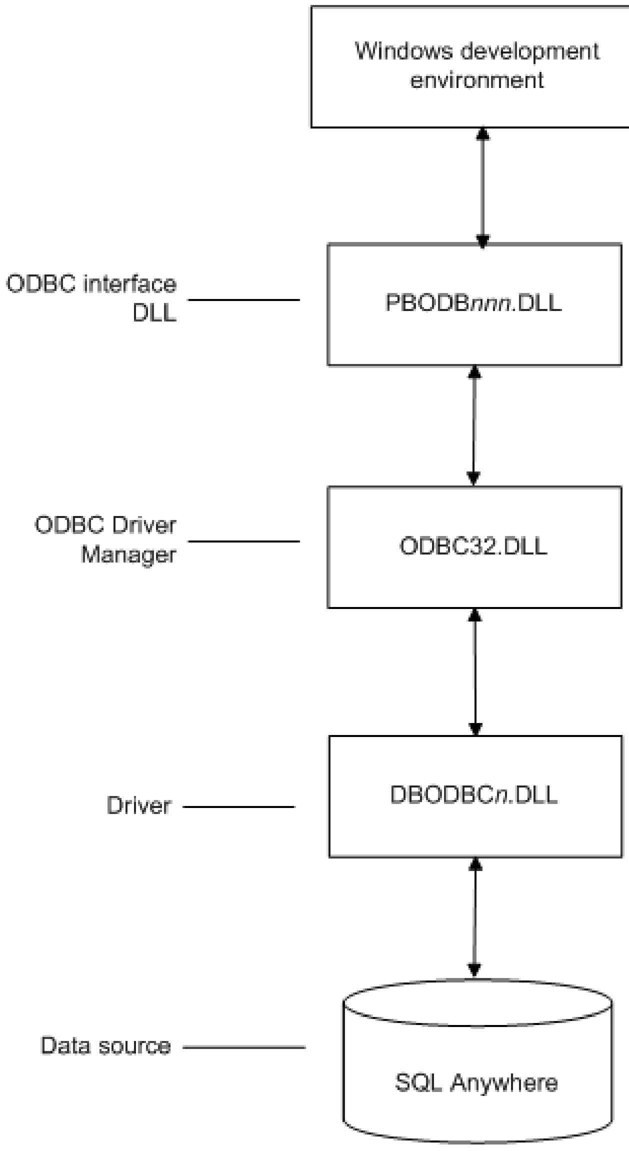
The SQL Anywhere ODBC driver supports connection to local and remote databases created with the following:

- PowerBuilder running on your computer
- SQL Anywhere version 17.x
- SQL Anywhere version 16.x
- SQL Anywhere version 12.x

Basic software components for SQL Anywhere

Figure 2-5 shows the basic software components required to connect to a SQL Anywhere data source in PowerBuilder.

Figure 2-5: Components of a SQL Anywhere connection



Preparing to use the SQL Anywhere data source

Before you define and connect to a SQL Anywhere data source in PowerBuilder, follow these steps to prepare the data source.

❖ **To prepare a SQL Anywhere data source:**

1. Make sure the database file for the SQL Anywhere data source already exists. You can create a new database by:

   - Launching the Create SQL Anywhere Database utility. You can access this utility from the Utilities folder for the ODBC interface in the Database profile or Database painter when PowerBuilder is running on your computer.

     This method creates a local SQL Anywhere database on your computer, and also creates the data source definition and database profile for this connection. (For instructions, see the *Users Guide*.)

   - Creating the database some other way, such as with PowerBuilder running on another user's computer or by using SQL Anywhere outside PowerBuilder. (For instructions, see the SQL Anywhere documentation.)

2. Make sure you have the log file associated with the SQL Anywhere database so that you can fully recover the database if it becomes corrupted.

   If the log file for the SQL Anywhere database does not exist, the SQL Anywhere database engine creates it. However, if you are copying or moving a database from another computer or directory, you should copy or move the log file with it.

Defining the SQL Anywhere data source

When you create a local SQL Anywhere database, PowerBuilder automatically creates the data source definition and database profile for you. Therefore, you need only use the following procedure to define a SQL Anywhere data source when you want to access a SQL Anywhere database not created using PowerBuilder on your computer.

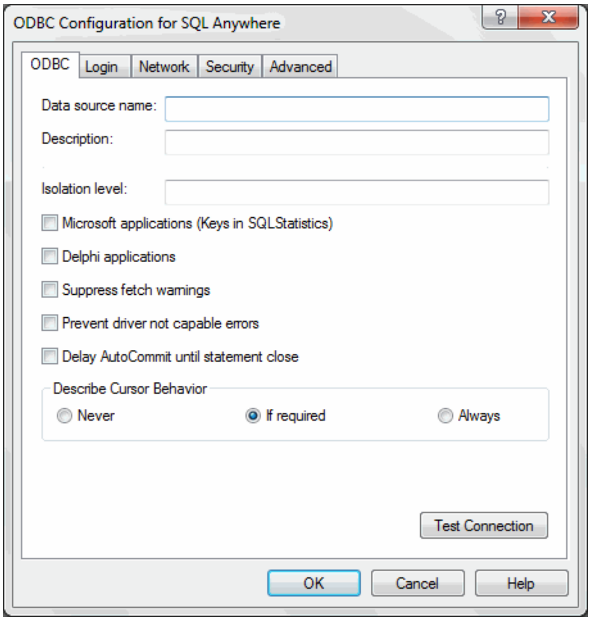
❖ **To define a SQL Anywhere data source for the SQL Anywhere driver:**

1. Select Create ODBC Data Source from the list of ODBC utilities in the Database Profiles dialog box or the Database painter.

2. Select User Data Source and click Next.

3 On the Create New Data Source page, select the SQL Anywhere driver and click Finish.

The ODBC Configuration for SQL Anywhere dialog box displays:


ODBC Configuration for SQL Anywhere

Data source name:

Description:

Isolation level:

- [ ] Microsoft applications (Keys in SQLStatistics)
- [ ] Delphi applications
- [ ] Suppress fetch warnings
- [ ] Prevent driver not capable errors
- [ ] Delay AutoCommit until statement close

Describe Cursor Behavior

Never [✓ If required] Always

4 You must supply the following values:

- Data source name on the ODBC tab page
- User ID and password on the Login tab page
- Database file on the Database tab page

Use the Help button to get information about fields in the dialog box.

5 (Optional) To select an ODBC translator to translate your data from one character set to another, click the Select button on the ODBC tab.

See Selecting an ODBC translator on page 29.

6 Click OK to save the data source definition.

Specifying a Start Line value

When the SQL Anywhere ODBC driver cannot find a running personal or network database server using the PATH variable and Database Name setting, it uses the commands specified in the Start Line field to start the database servers.

Specify one of the following commands in the Start Line field on the Database tab page, where *n* is the version of SQL Anywhere you are using.

Specify this command	To
dbeng*n*.exe	Start the personal database server and the database specified in the Database File box
rteng*n*.exe	Start the restricted runtime database server and the database specified in the Database File box

For information on completing the ODBC Configuration For SQL Anywhere dialog box, see the SQL Anywhere documentation.

Support for Transact-SQL special timestamp columns

When you work with a SQL Anywhere table in the DataWindow®, Data Pipeline, or Database painter, the default behavior is to treat any column named timestamp as a SQL Anywhere Transact-SQL® special timestamp column.

Creating special timestamp columns

You can create a Transact-SQL special timestamp column in a SQL Anywhere table.

❖ To create a Transact-SQL special timestamp column in a SQL Anywhere table in PowerBuilder:

1. Give the name timestamp to any column having a timestamp datatype that you want treated as a Transact-SQL special timestamp column. Do this in one of the following ways:

   - In the painter – Select timestamp as the column name. (For instructions, see the *Users Guide*.)

   - In a SQL CREATE TABLE statement – Follow the CREATE TABLE example next.

2. Specify *timestamp* as the default value for the column. Do this in one of the following ways:

   - In the painter – Select timestamp as the default value for the column. (For instructions, see the *Users Guide*.)

   - In a SQL CREATE TABLE statement – Follow the CREATE TABLE example next.

3 If you are working with the table in the Data Pipeline painter, select the initial value exclude for the special timestamp column from the drop-down list in the Initial Value column of the workspace.

You must select exclude as the initial value to exclude the special timestamp column from INSERT or UPDATE statements.

For instructions, see the *Users Guide*.

CREATE TABLE example

The following CREATE TABLE statement defines a SQL Anywhere table named timesheet containing three columns: employee_ID (integer datatype), hours (decimal datatype), and timestamp (timestamp datatype and timestamp default value):

```
CREATE TABLE timesheet (
   employee_ID INTEGER,
   hours DECIMAL,
   timestamp TIMESTAMP default timestamp )
```

Not using special timestamp columns

If you want to change the default behavior, you can specify that PowerBuilder *not* treat SQL Anywhere columns named *timestamp* as Transact-SQL special timestamp columns.

❖ **To specify that PowerBuilder *not* treat columns named *timestamp* as a Transact-SQL special timestamp column:**

- Edit the SAP SQL Anywhere section of the PBODB170 initialization file to change the value of SQLSrvrTSName from `'Yes'` to `'No'`.

  After making changes in the initialization file, you must reconnect to the database to have them take effect. See the Appendix, Adding Functions to the PBODB170 Initialization File.

What to do next

For instructions on connecting to the ODBC data source, see Connecting to a database on page 169.

CHAPTER 3 Using the JDBC Interface

About this chapter

This chapter describes the JDBC interface and explains how to prepare to use this interface and how to define the JDBC database profile.

Contents

For more information

For more detailed information about JDBC, go to the Java Web site at http://java.sun.com/products/jdbc/overview.html.

About the JDBC interface

You can access a wide variety of databases through JDBC in PowerBuilder. This section describes what you need to know to use JDBC connections to access your data in PowerBuilder.

What is JDBC?

The JDBC API

Java Database Connectivity (JDBC) is a standard application programming interface (API) that allows a Java application to access any database that supports Structured Query Language (SQL) as its standard data access language.

The JDBC API includes classes for common SQL database activities that allow you to open connections to databases, execute SQL commands, and process results. Consequently, Java programs have the capability to use the familiar SQL programming model of issuing SQL statements and processing the resulting data. The JDBC classes are included in Java 1.1+ and Java 2 as the java.sql package.

The JDBC API defines the following:

- A library of JDBC function calls that connect to a database, execute SQL statements, and retrieve results
- A standard way to connect and log in to a DBMS
- SQL syntax based on the X/Open SQL Call Level Interface or X/Open and SQL Access Group (SAG) CAE specification (1992)
- A standard representation for datatypes
- A standard set of error codes

How JDBC APIs are implemented

JDBC API implementations fall into two broad categories: those that communicate with an existing ODBC driver (a JDBC-ODBC bridge) and those that communicate with a native database API (a JDBC driver that converts JDBC calls into the communications protocol used by the specific database vendor). The PowerBuilder implementation of the JDBC interface can be used to connect to any database for which a JDBC-compliant driver exists.

The PowerBuilder JDB interface

A Java Virtual Machine (JVM) is required to interpret and execute the bytecode of a Java program. The PowerBuilder JDB interface supports the Sun Java Runtime Environment (JRE) versions 1.2 and later.

Using the JDBC interface

You can use the JDBC interface to develop the client/server applications. If a client is already running a JVM (in a running Web browser or inside the operating system), the use of the JDBC interface to access a database does not require the client-side installation and administration of a database driver, which is required when using ODBC.

Components of a JDBC connection

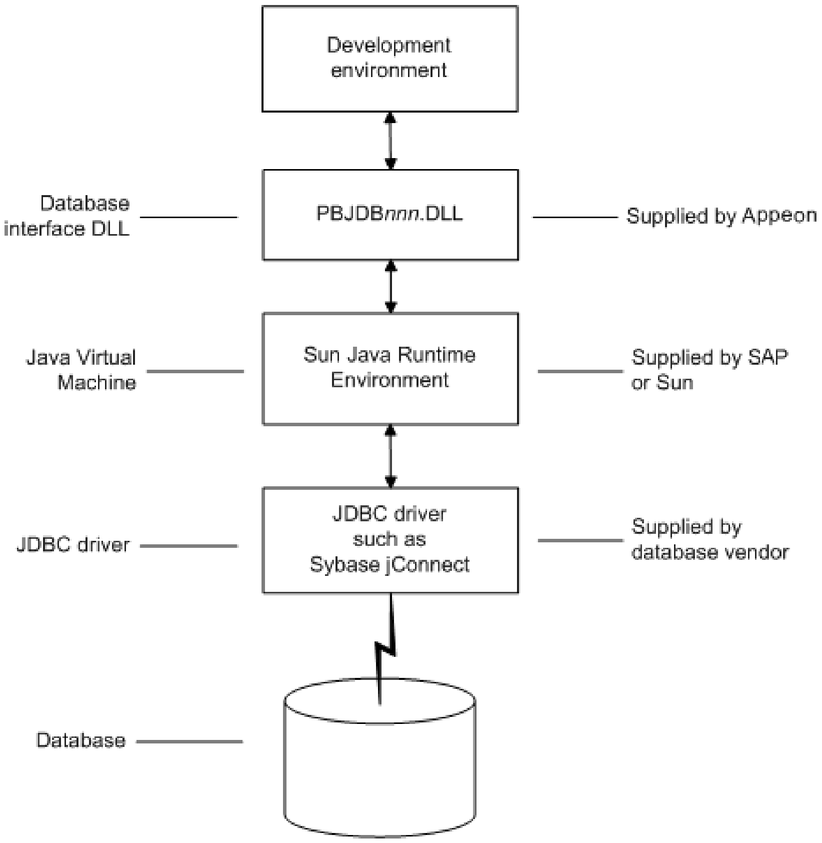
How a JDBC connection is made

In PowerBuilder when you access a database through the JDBC interface, your connection goes through several layers before reaching the database. It is important to understand that each layer represents a separate component of the connection, and that each component might come from a different vendor.

Because JDBC is a standard API, PowerBuilder uses the same interface to access every JDBC-compliant database driver.

Figure 3-1 shows the general components of a JDBC connection.

Figure 3-1: Components of a JDBC connection



The JDBC DLL

PowerBuilder provides the *pbjdb170.dll*. This DLL runs with the Sun Java Runtime Environment (JRE) versions 1.1 and later.

PowerBuilder Java package

PowerBuilder includes a small package of Java classes that gives the JDBC interface the level of error-checking and efficiency (SQLException catching) found in other PowerBuilder interfaces. The package is called *pbjdbc12170.jar* and is found in *Appeon\Shared\PowerBuilder*.

The Java Virtual Machine

The Java Virtual Machine (JVM) is a component of Java development software. When you install PowerBuilder, the Sun Java Development Kit (JDK), including the Java Runtime Environment (JRE), is installed on your system in *Appeon\Shared\PowerBuilder*. For PowerBuilder 17.0, JDK 1.6 is installed. This version of the JVM is started when you use a JDBC connection or any other process that requires a JVM and is used throughout the PowerBuilder session.

If you need to use a different JVM, see the instructions in Preparing to use the JDBC interface on page 41. For more information about how the JVM is started, see the chapter on deploying your application in *Application Techniques*.

The JDBC drivers

The JDBC interface can communicate with any JDBC-compliant driver including SAP jConnect™ for JDBC (available with SAP ASE, IQ, and SA database clients) and the Oracle and IBM Informix JDBC drivers. These drivers are native-protocol, all-Java drivers—that is, they convert JDBC calls into the SQL syntax supported by the databases.

Accessing Unicode data

Using the ODBC interface, PowerBuilder can connect, save, and retrieve data in both ANSI/DBCS and Unicode databases but does not convert data between Unicode and ANSI/DBCS. When character data or command text is sent to the database, PowerBuilder sends a Unicode string. The driver must guarantee that the data is saved as Unicode data correctly. When PowerBuilder retrieves character data, it assumes the data is Unicode.

A Unicode database is a database whose character set is set to a Unicode format, such as UTF-8, UTF-16, UCS-2, or UCS-4. All data must be in Unicode format, and any data saved to the database must be converted to Unicode data implicitly or explicitly.

A database that uses ANSI (or DBCS) as its character set might use special datatypes to store Unicode data. Columns with these datatypes can store *only* Unicode data. Any data saved into such a column must be converted to Unicode explicitly. This conversion must be handled by the database server or client.

JDBC registry entries

When you access data through the PowerBuilder JDBC interface, PowerBuilder uses an internal registry to maintain definitions of SQL syntax, DBMS-specific function calls, and default DBParm parameter settings for the back-end DBMS. This internal registry currently includes subentries for SQL Anywhere, Adaptive Server Enterprise, and Oracle databases.

In most cases you do not need to modify the JDBC entries. However, if you do need to customize the existing entries or add new entries, you can make changes to the system registry by editing the registry directly or executing a registry file. Changes you introduce in the system registry override the PowerBuilder internal registry entries. See the *egreg.txt* file in *Appeon\Shared\PowerBuilder* for an example of a registry file you could execute to change entry settings.

Supported versions for JDBC

The PowerBuilder JDBC interface uses the *pbjdb170.dll* to access a database through a JDBC driver.

To use the JDBC interface to access the jConnect driver, use jConnect Version 4.2 or higher. For information on jConnect, see your SAP documentation.

To use the JDBC interface to access the Oracle JDBC driver, use Oracle 8 JDBC driver Version 8.0.4 or higher. For information on the Oracle JDBC driver, see your Oracle documentation.

Supported JDBC datatypes

Like ODBC, the JDBC interface compiles, sorts, presents, and uses a list of datatypes that are native to the back-end database to emulate as much as possible the behavior of a native interface.

Preparing to use the JDBC interface

Before you define the interface and connect to a database through the JDBC interface, follow these steps to prepare the database for use:

1. Configure the database server for its JDBC connection and install its JDBC-compliant driver and network software.
2. Install the JDBC driver.
3. Set or verify the settings in the CLASSPATH environment variable and the Java tab of the System Options dialog box.

Step 1: Configure the database server

You must configure the database server to make JDBC connections as well as install the JDBC driver and network software.

❖ **To configure the database server for its JDBC connection:**

1. Make sure the database server is configured to make JDBC connections. For configuration instructions, see your database vendor's documentation.

2. Make sure the appropriate JDBC driver software is installed and running on the database server.

   The driver vendor's documentation should provide the driver name, URL format, and any driver-specific properties you need to specify in the database profile. For notes about the jConnect driver, see Configuring the jConnect driver on page 42.

3. Make sure the required network software (such as TCP/IP) is installed and running on your computer and is properly configured so that you can connect to the database server at your site.

   You must install the network communication driver that supports the network protocol and operating system platform you are using.

   For installation and configuration instructions, see your network or database administrator.

Step 2: Install the JDBC driver

In the PowerBuilder Setup program, select the Typical install, or select the Custom install and select the JDBC driver.

Step 3: Verify or set the settings in the CLASSPATH variable and Java tab

Verify that the settings in the PATH and CLASSPATH environment variables or the Classpaths list on the Java tab of the PowerBuilder System Options dialog box point to the appropriate, fully qualified file names, or set them.

If you are using the JDK installed with PowerBuilder, you do not need to make any changes to these environment variables.

If you are using JDK 1.2 or later, you do not need to include any Sun Java VM packages in your CLASSPATH variable, but your PATH environment variable must include an entry for the Sun Java VM library, *jvm.dll* (for example, *path\JDK15\JRE\bin\client*).

Configuring the jConnect driver

If you are using the SAP jConnect driver, make sure to complete the required configuration steps such as installing the JDBC stored procedures in Adaptive Server databases. Also, verify that the CLASSPATH environment variable on your computer or the Classpaths list on the Java tab of the PowerBuilder System Options dialog box includes an entry pointing to the location of the jConnect driver.

For example, if you are using jConnect 6.05, you should include an entry similar to the following:

```
C:\Program Files\SAP\jConnect-6.05\classes\jconn3.jar
```

For more information about configuring jConnect, see the jConnect for JDBC documentation.

Defining the JDBC interface

Defining the profile

To define a connection through the JDBC interface, you must create a database profile by supplying values for at least the basic connection parameters in the Database Profile Setup - JDBC dialog box. You can then select this profile at any time to connect to your database in the development environment.

For information on how to define a database profile, see Using database profiles on page 6.

Specifying connection parameters

To provide maximum flexibility (as provided in the JDBC API), the JDBC interface supports database connections made with different combinations of connection parameters:

- **Driver name, URL, and Properties** You should specify values for this combination of connection parameters if you need to define driver-specific properties. When properties are defined, you *must* also define the user ID and password in the properties field.

  For example, when connecting to the jConnect driver, enter the following values in the Driver-Specific Properties field:

  ```
  SQLINITSTRING=set TextSize 32000;
  user=system;password=manager;
  ```

- **Driver name, URL, User ID, and Password** You should specify values for this combination of connection parameters if you do not need to define any driver-specific properties.

  ```
  Driver Name: com.sybase.jdbc3.jdbc.SybDriver
  URL: jdbc:sybase:Tds:localhost:2638
  Login ID:    dba
  Password:    sql
  ```

- **Driver name and URL** You should specify values for this combination of connection parameters when the user ID and password are included as part of the URL.

  For example, when connecting to the Oracle JDBC driver, the URL can include the user ID and password:

  ```
  jdbc:oracle:thin:userid/password@host:port:dbname
  ```

Specifying properties when connecting to jConnect

If you plan to use the blob datatype in PowerBuilder, you should be aware that jConnect imposes a restriction on blob size. Consequently, before you make your database connection from PowerBuilder, you might want to reset the blob size to a value greater than the maximum size you plan to use.

To set blob size, define the jConnect property SQLINITSTRING in the Driver-Specific Properties box on the Connection page. The SQLINITSTRING property is used to define commands to be passed to the back-end database server:

```
SQLINITSTRING=set TextSize 32000;
```

Remember that if you define a property in the Driver-Specific Properties box, you must also define the user ID and password in this box.

CHAPTER 4 Using the OLE DB Interface

About this chapter

This chapter describes the OLE DB interface and explains how to prepare to use this interface and how to define the OLE DB database profile.

Contents

For more information

This chapter gives general information about using the OLE DB interface. For more detailed information:

- See the *OLE DB Programmer's Guide* in the Microsoft MSDN library at http://msdn.microsoft.com/en-us/library/ms713643.aspx.
- Use the online Help provided by the data provider vendor.
- Check to see if there is a technical document that describes how to connect to your OLE DB data provider.

About the OLE DB interface

You can access a wide variety of data through OLE DB data providers in PowerBuilder. This section describes what you need to know to use OLE DB connections to access your data in PowerBuilder.

Supported OLE DB data providers

For a complete list of the OLE DB data providers supplied with PowerBuilder and the data they access, see "Supported Database Interfaces" in the online Help.

What is OLE DB?

OLE DB API

OLE DB is a standard application programming interface (API) developed by Microsoft. It is a component of Microsoft's Data Access Components software. OLE DB allows an application to access a variety of data for which OLE DB data providers exist. It provides an application with uniform access to data stored in diverse formats, such as indexed-sequential files like Btrieve, personal databases like Paradox, productivity tools such as spreadsheets and electronic mail, and SQL-based DBMSs.

The OLE DB interface supports direct connections to SQL-based databases.

Accessing data through OLE DB

Applications like PowerBuilder that provide an OLE DB interface can access data for which an OLE DB data provider exists. An **OLE DB data provider** is a dynamic link library (DLL) that implements OLE DB function calls to access a particular data source.

The PowerBuilder OLE DB interface can connect to any OLE DB data provider that supports the OLE DB object interfaces listed in Table 4-1. An OLE DB data provider must support these interfaces in order to adhere to the Microsoft OLE DB 2.0 specification.

Table 4-1: Required OLE DB interfaces

IAccessor	IDBInitialize
IColumnsInfo	IDBProperties
ICommand	IOpenRowset
ICommandProperties	IRowset
ICommandText	IRowsetInfo
IDBCreateCommand	IDBSchemaRowset
IDBCreateSession	ISourcesRowset

In addition to the required OLE DB interfaces, PowerBuilder also uses the OLE DB interfaces listed in Table 4-2 to provide further functionality.

Table 4-2: Additional OLE DB interfaces

OLE DB interface	Use in PowerBuilder
ICommandPrepare	Preparing commands and retrieving column information.
IDBInfo	Querying the data provider for its properties. If this interface is not supported, database connections might fail.
IDBCommandWithParameters	Querying the data provider for parameters.
IErrorInfo	Providing error information.
IErrorRecords	Providing error information.

OLE DB interface	Use in PowerBuilder
IIndexDefinition	Creating indexes for the extended attribute system tables. Also creating indexes in the Database painter. If this interface is not supported, PowerBuilder looks for index definition syntax in the *pbodb170.ini* file.
IMultipleResults	Providing information.
IRowsetChange	Populating the extended attribute system tables when they are created. Also, for updating blobs.
IRowsetUpdate	Creating the extended attribute system tables.
ISQLErrorInfo	Providing error information.
ISupportErrorInfo	Providing error information.
ITableDefinition	Creating the extended attribute system tables and also for creating tables in the Database painter. If this interface is not supported, the following behavior results: • PowerBuilder looks for table definition syntax in the *pbodb170.ini* file • PowerBuilder catalog tables cannot be used • DDL and DML operations, like modifying columns or editing data in the database painter, do not function properly
ITransactionLocal	Supporting transactions. If this interface is not supported, PowerBuilder defaults to AutoCommit mode.

Accessing Unicode data

Using the OLE DB interface, PowerBuilder can connect, save, and retrieve data in both ANSI/DBCS and Unicode databases but does not convert data between Unicode and ANSI/DBCS. When character data or command text is sent to the database, PowerBuilder sends a Unicode string. The data provider must guarantee that the data is saved as Unicode data correctly. When PowerBuilder retrieves character data, it assumes the data is Unicode.

A Unicode database is a database whose character set is set to a Unicode format, such as UTF-8, UTF-16, UCS-2, or UCS-4. All data must be in Unicode format, and any data saved to the database must be converted to Unicode data implicitly or explicitly.

A database that uses ANSI (or DBCS) as its character set might use special datatypes to store Unicode data. Columns with these datatypes can store *only* Unicode data. Any data saved into such a column must be converted to Unicode explicitly. This conversion must be handled by the database server or client.

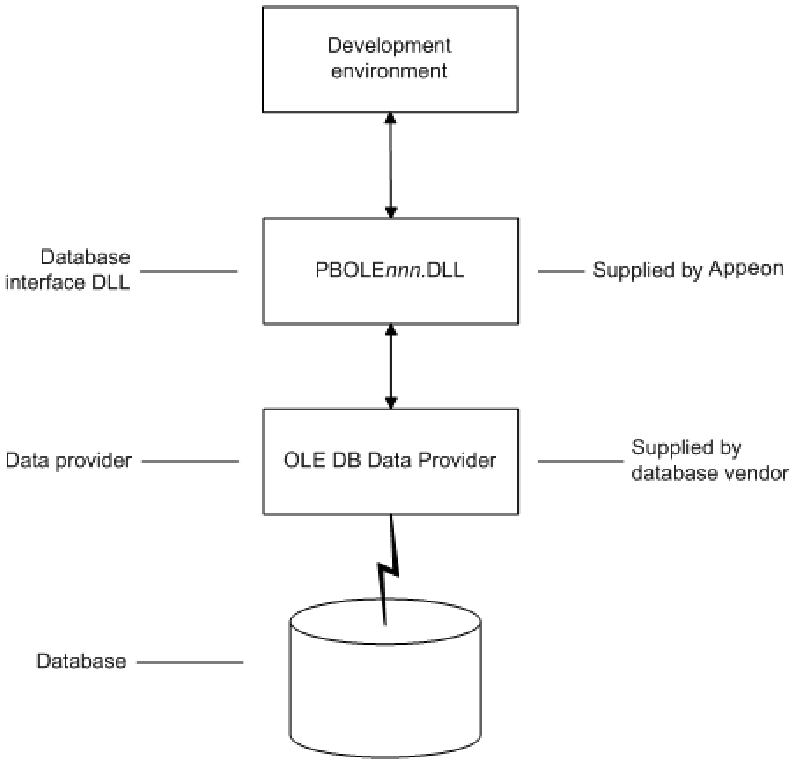
Components of an OLE DB connection

When you access an OLE DB data provider in PowerBuilder, your connection goes through several layers before reaching the data provider. It is important to understand that each layer represents a separate component of the connection, and that each component might come from a different vendor.

Because OLE DB is a standard API, PowerBuilder uses the same interface to access every OLE DB data provider. As long as an OLE DB data provider supports the object interfaces required by PowerBuilder, PowerBuilder can access it through the OLE DB interface.

Figure 4-1 shows the general components of a OLE DB connection.

Figure 4-1: Components of an OLE DB connection



Obtaining OLE DB data providers

PowerBuilder Enterprise lets you access data with *any* OLE DB data provider obtained from a vendor other than SAP if that data provider supports the OLE DB object interfaces required by PowerBuilder. In most cases, these drivers work with PowerBuilder. However, SAP might not have tested the drivers to verify this.

Supported versions for OLE DB

The OLE DB interface uses a DLL named *PBOLE170.DLL* to access a database through an OLE DB data provider.

Required OLE DB version
To use the OLE DB interface to access an OLE DB database, you must connect through an OLE DB data provider that supports OLE DB version 2.0 or later. For information on OLE DB specifications, see the Microsoft documentation at http://msdn.microsoft.com/en-us/library/default.aspx.

Preparing to use the OLE DB interface

Before you define the interface and connect to a data provider through OLE DB:

1. Install and configure the database server, network, and client software.
2. Install the OLE DB interface and the OLE DB data provider that accesses your data source.
3. Install Microsoft's Data Access Components software on your machine.
4. If required, define the OLE DB data source.

Step 1: Install and configure the data server

You must install and configure the database server and install the network software and client software.

❖ To install and configure the database server, network, and client software:

1. Make sure the appropriate database software is installed and running on its server.

   You must obtain the database server software from your database vendor. For installation instructions, see your database vendor's documentation.

2. Make sure the required network software (such as TCP/IP) is installed and running on your computer and is properly configured so that you can connect to the data server at your site. You must install the network communication driver that supports the network protocol and operating system platform you are using.

   For installation and configuration instructions, see your network or data source administrator.

3. If required, install the appropriate client software on each client computer on which PowerBuilder is installed.

Client software requirements
To determine client software requirements, see your database vendor's documentation.

Step 2: Install the OLE DB interface and data provider

In the PowerBuilder Setup program, select the Custom install and select the OLE DB provider that accesses your database. You can install one or more of the OLE DB data providers shipped with PowerBuilder, or you can install data providers from another vendor later.

Step 3: Install the Microsoft Data Access Components software

The PowerBuilder OLE DB interface requires the functionality of the Microsoft Data Access Components (MDAC) version 2.8 or higher software.

To check the version of MDAC on your computer, you can download and run the MDAC Component Checker utility from the MDAC Downloads page at http://msdn.microsoft.com/en-us/data/aa937730.aspx.

OLE DB data providers installed with MDAC
Several Microsoft OLE DB data providers are automatically installed with MDAC, including the providers for SQL Server (SQLOLEDB) and ODBC (MSDASQL).

Step 4: Define the OLE DB data source

Once the OLE DB data provider is installed, you might have to define the OLE DB data source the data provider will access. How you define the data source depends on the OLE DB data provider you are using and the vendor who provided it.

If you are connecting to an ODBC data provider (such as Microsoft's OLE DB Provider for ODBC), you must define the ODBC data source as you would if you were using a direct ODBC connection. To define an ODBC data source, use Microsoft's ODBC Data Source Administrator. You can access this utility from the Control Panel in Windows or from the Database painter or Database Profile Setup dialog box in PowerBuilder.

Defining the OLE DB interface

Using the OLE DB Database Profile Setup

To define a connection through the OLE DB interface, you must create a database profile by supplying values for at least the basic connection parameters in the Database Profile Setup – OLE DB dialog box. You can then select this profile anytime to connect to your data in the development environment.

For information on how to define a database profile, see Using database profiles on page 6.

Specifying connection parameters

You must supply values for the Provider and Data Source connection parameters. Select a data provider from the list of installed data providers in the Provider drop-down list. The Data Source value varies depending on the type of data source connection you are making. For example:

- If you are using Microsoft's OLE DB Provider for ODBC to connect to the Demo Database, you select MSDASQL as the Provider value and enter the actual ODBC data source name (for example, Demo Database) as the Data Source value.
- If you are using Microsoft's OLE DB Provider for SQL Server, you select SQLOLEDB as the Provider value and enter the actual server name as the Data Source value. You must also use the Extended Properties field to provide the database name (for example, Database=Pubs) since you can have multiple instances of a database.

Using the Data Link API

The Data Link option allows you to access Microsoft's Data Link API, which allows you to define a file or use an existing file that contains your OLE DB connection information. A Data Link file is identified with the suffix *.udl*. If you use a Data Link file to connect to your data source, all other settings you make in the OLE DB Database Profile Setup dialog box are ignored.

To launch this option, select the File Name check box on the Connection tab and double-click on the button next to the File Name box. (You can also launch the Data Link API in the Database painter by double-clicking on the Manage Data Links utility included with the OLE DB interface in the list of Installed Database Interfaces.)

For more information on using the Data Link API, see the *OLE DB Programmer's Guide* in the Microsoft MSDN library at http://msdn.microsoft.com/en-us/library/ms713643.aspx.

CHAPTER 5 Using the ADO.NET Interface

About this chapter

This chapter describes the ADO.NET interface and explains how to prepare to use this interface and how to define an ADO.NET database profile.

Contents

For more information

This chapter gives general information about using the ADO.NET interface. For more detailed information:

- See the Data Access and .NET development sections in the Microsoft MSDN library at http://msdn.microsoft.com/en-us/data/default.aspx.
- Use the online Help provided by the data provider vendor.
- Check to see if there is a technical document that describes how to connect to your ADO.NET data provider.

About ADO.NET

ADO.NET is a set of technologies that provides native access to data in the Microsoft .NET Framework. It is designed to support an n-tier programming environment and to handle a disconnected data architecture. ADO.NET is tightly integrated with XML and uses a common data representation that can combine data from disparate sources, including XML.

One of the major components of ADO.NET is the .NET Framework data provider, which connects to a database, executes commands, and retrieves results.

Microsoft provides .NET Framework data providers for SQL Server and OLE DB with the .NET Framework, and data providers for ODBC and Oracle can be downloaded from the Microsoft Web site. You can also obtain .NET Framework data providers from other vendors, such as the .NET Framework Data Provider for Adaptive Server Enterprise from SAP.

To connect to a database using the PowerBuilder ADO.NET database interface, you must use a .NET Framework data provider.

Accessing Unicode data

Using the ADO.NET interface, PowerBuilder can connect, save, and retrieve data in both ANSI/DBCS and Unicode databases but does not convert data between Unicode and ANSI/DBCS. When character data or command text is sent to the database, PowerBuilder sends a Unicode string. The data provider must guarantee that the data is saved as Unicode data correctly. When PowerBuilder retrieves character data, it assumes the data is Unicode.

A Unicode database is a database whose character set is set to a Unicode format, such as UTF-8, UTF-16, UCS-2, or UCS-4. All data must be in Unicode format, and any data saved to the database must be converted to Unicode data implicitly or explicitly.

A database that uses ANSI (or DBCS) as its character set might use special datatypes to store Unicode data. Columns with these datatypes can store *only* Unicode data. Any data saved into such a column must be converted to Unicode explicitly. This conversion must be handled by the database server or client.

About the PowerBuilder ADO.NET database interface

You can use the PowerBuilder ADO.NET database interface to connect to a data source such as Adaptive Server® Enterprise, Oracle, and Microsoft SQL Server, as well as to data sources exposed through OLE DB and XML, in much the same way as you use the PowerBuilder ODBC and OLE DB database interfaces.

Performance
You might experience better performance if you use a native database interface. The primary purpose of the ADO.NET interface is to support shared connections with other database constructs such as the .NET DataGrid in SAP DataWindow .NET.

Components of an ADO.NET connection

When you access a database using ADO.NET in PowerBuilder, your connection goes through several layers before reaching the database. It is important to understand that each layer represents a separate component of the connection, and that components might come from different vendors.

The PowerBuilder ADO.NET interface consists of a driver (*pbado170.dll*) and a server (either *Sybase.PowerBuilder.Db.dll* or *Sybase.PowerBuilder.DbExt.dll*). The server has dependencies on a file called *pbrth170.dll*. These DLLs must be deployed with an application that connects to a database using ADO.NET. For Oracle 10*g* or Adaptive Server 15 or later, use *Sybase.PowerBuilder.DbExt.dll*. For earlier versions and other DBMSs, use *Sybase.PowerBuilder.Db.dll*.

The DataWindow .NET database interface for ADO.NET supports the ADO.NET data providers listed in Table 5-1.

Table 5-1: Supported ADO.NET data providers

Data Provider	Namespace
.NET Framework Data Provider for OLE DB	System.Data.OleDb
.NET Framework Data Provider for SQL Server	System.Data.SqlClient
Oracle Data Provider for .NET (ODP.NET)	Oracle.DataAccess.Client
SAP ADO.NET Data Provider for Adaptive Server Enterprise (ASE)	SAP.Data.AseClient

Additional .NET Framework data providers may be supported in future releases. Please see the release bulletin for the latest information.

Figure 5-1 shows the general components of an ADO.NET connection using the OLE DB .NET Framework data provider.

Figure 5-1: Components of an ADO.NET OLE DB connection

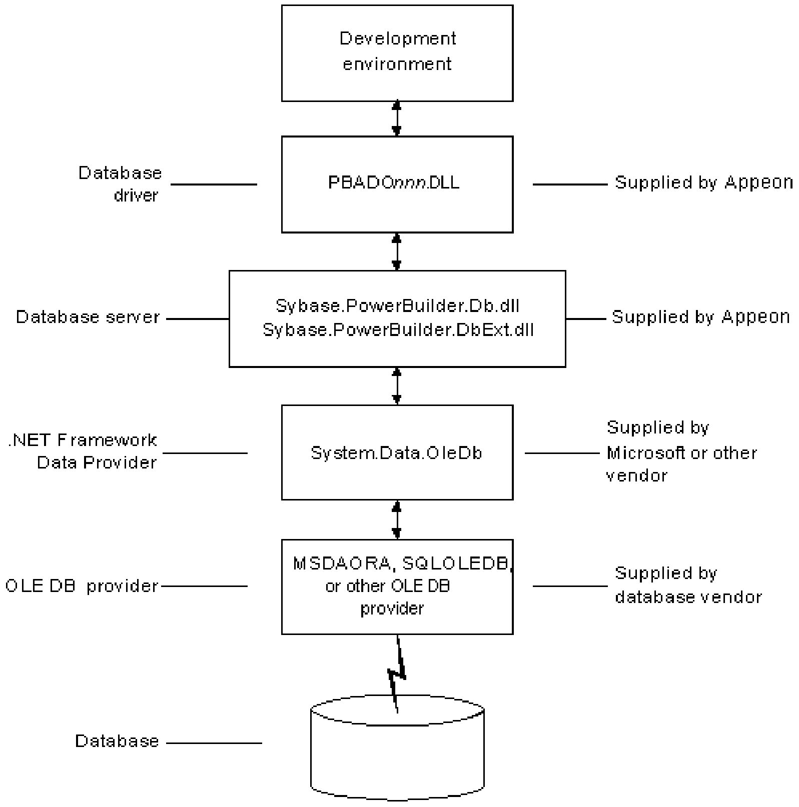
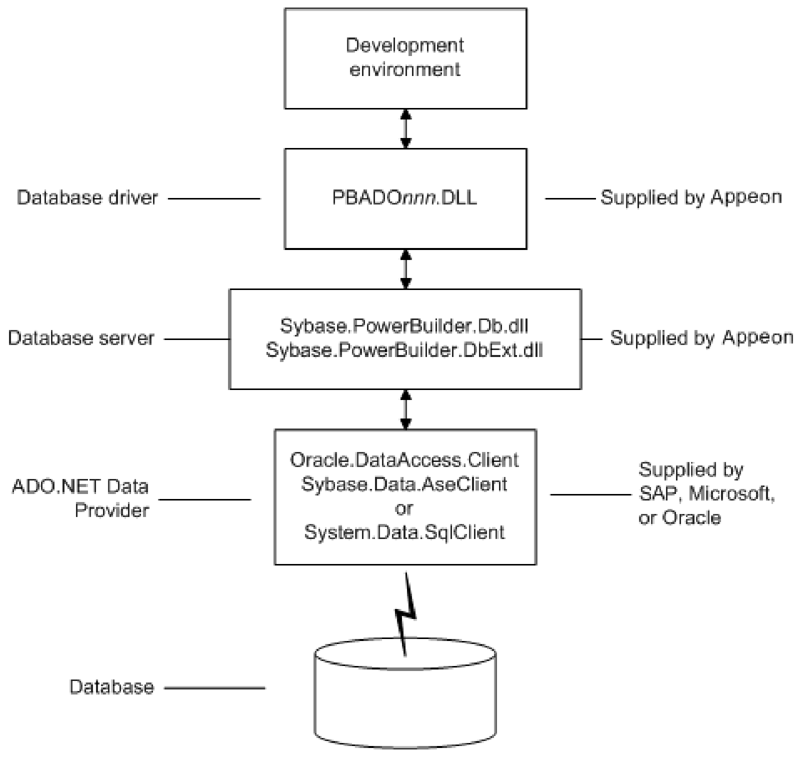

Development environment
Database driver
PBADOnnn.DLL
Supplied by Appeon
Database server
Sybase.PowerBuilder.Db.dll
Sybase.PowerBuilder.DbExt.dll
Supplied by Appeon
.NET Framework Data Provider
System.Data.OleDb
Supplied by Microsoft or other vendor
OLE DB provider
MSDAORA, SQLOLEDB, or other OLE DB provider
Supplied by database vendor
Database

Figure 5-2 shows the general components of an ADO.NET connection using a native ADO.NET data provider.

Figure 5-2: Components of a native ADO.NET connection



OLE DB data providers

When you use the .NET Framework data provider for OLE DB, you connect to a database through an OLE DB data provider, such as Microsoft's SQLOLEDB or MSDAORA or a data provider from another vendor.

The .NET Framework Data Provider for OLE DB does not work with the MSDASQL provider for ODBC, and it does not support OLE DB version 2.5 interfaces.

You can use any OLE DB data provider that supports the OLE DB interfaces listed in Table 5-2 with the OLE DB .NET Framework data provider. For more information about supported providers, see the topic on .NET Framework data providers in the Microsoft *.NET Framework Developer's Guide*.

The PowerBuilder ADO.NET interface supports connection to SQL Anywhere, Adaptive Server Enterprise, Microsoft SQL Server, Oracle, Informix, and Microsoft Access with the OLE DB .NET Framework data provider.

After you install the data provider, you might need to define a data source for it.

Table 5-2: Required interface support for OLE DB data providers

OLE DB object	Required interfaces
OLE DB Services	IDataInitialize
DataSource	IDBInitialize IDBCreateSession IDBProperties IPersist
Session	ISessionProperties IOpenRowset
Command	ICommandText ICommandProperties
MultipleResults	IMultipleResults
RowSet	IRowset IAccessor IColumnsInfo IRowsetInfo (only required if DBTYPE_HCHAPTER is supported)
Error	IErrorInfo IErrorRecords

Preparing to use the ADO.NET interface

Before you define the interface and connect to a database using ADO.NET:

1. Install and configure the database server, network, and client software.
2. Install the ADO.NET interface.
3. Install Microsoft's Data Access Components version 2.6 or higher software on your machine.

Step 1: Install and configure the data server

You must install and configure the database server and install the network software and client software.

❖ To install and configure the database server, network, and client software:

1. Make sure the appropriate database software is installed and running on its server.

   You must obtain the database server software from your database vendor. For installation instructions, see your database vendor's documentation.

2. Make sure the required network software (such as TCP/IP) is installed and running on your computer and is properly configured so that you can connect to the data server at your site. You must install the network communication driver that supports the network protocol and operating system platform you are using.

   For installation and configuration instructions, see your network or data source administrator.

3. If required, install the appropriate client software on each client computer on which PowerBuilder is installed.

Client software requirements
To determine client software requirements, see your database vendor's documentation.

Step 2: Install the ADO.NET interface

In the PowerBuilder Setup program, select the Custom install and select the ADO.NET database interface.

Step 3: Install the Microsoft Data Access Components software

The PowerBuilder ADO.NET interface requires the functionality of the Microsoft Data Access Components (MDAC) version 2.8 or higher software.

To check the version of MDAC on your computer, you can download and run the MDAC Component Checker utility from the MDAC Downloads page at http://msdn.microsoft.com/en-us/data/aa937730.aspx.

OLE DB data providers installed with MDAC
Several Microsoft OLE DB data providers are automatically installed with MDAC, including the providers for SQL Server (SQLOLEDB) and ODBC (MSDASQL).

Defining the ADO.NET interface

Using the ADO.NET Database Profile Setup

To define a connection using the ADO.NET interface, you must create a database profile by supplying values for at least the basic connection parameters in the Database Profile Setup – ADO.NET dialog box. You can then select this profile at any time to connect to your data in PowerBuilder.

For information on how to define a database profile, see Using database profiles on page 6.

Specifying connection parameters

You must supply a value for the Namespace and DataSource connection parameters and for the User ID and Password. When you use the System.Data.OleDb namespace, you must also select a data provider from the list of installed data providers in the Provider drop-down list.

The Data Source value varies depending on the type of data source connection you are making. For example, if you are using Microsoft's OLE DB Provider for SQL Server, you select SQLOLEDB as the Provider value and enter the actual server name as the Data Source value. In the case of Microsoft SQL Server, you must also use the Extended Properties field to provide the database name (for example, Database=Pubs) since you can have multiple instances of a database.

Using the Data Link API with OLE DB

The Data Link option allows you to access Microsoft's Data Link API, which allows you to define a file or use an existing file that contains your OLE DB connection information. A Data Link file is identified with the suffix *.udl*.

To launch this option, select the File Name check box on the Connection page and double-click the button next to the File Name box. (You can also launch the Data Link API in the Database painter by double-clicking the Manage Data Links utility included with the OLE DB interface in the list of Installed Database Interfaces.)

For more information on using the Data Link API, see Microsoft's Universal Data Access Web site at http://msdn.microsoft.com/en-us/data/default.aspx.

Using a Data Link file versus setting the database parameters
If you use a Data Link file to connect to your data source, all other database-specific settings you make in the ADO.NET Database Profile Setup dialog box are ignored.

Getting identity column values

You can use the standard `select @@identity` syntax to obtain the value of an identity column. You can also use an alternative syntax, such as `select scope_identity()`, by adding sections to a .NET configuration file for your application.

Setting up a dbConfiguration section in a configuration file

The following example shows the general structure of a configuration file with a database configuration section and one custom configuration section:

```
<configuration>
   <configSections>
     <sectionGroup name="dbConfiguration">
        <section name="mycustomconfig"
         type="Sybase.PowerBuilder.Db.DbConfiguration,
         Sybase.PowerBuilder.Db"
        />
      </sectionGroup>
   </configSections>

   <dbConfiguration>
      <mycustomconfig dbParm="optional_value"
       getIdentity="optional_syntax"
      />
   </dbConfiguration>
</configuration>
```

❖ **To add a database configuration section to a .NET configuration file:**

1. In the <configSections> section of the configuration file, add a <sectionGroup> element with the name “dbConfiguration”. This name is case sensitive.

   <configSections> must appear at the beginning of the configuration file, before the <runtime> section if any.

2. In the dbConfiguration <sectionGroup> element, add one of more <section> elements.

   For each section, specify a name of your choice and a type. The type is the strong name of the assembly used to parse this section of the configuration file.

3. Close the <section> and <configSections> elements and add a <dbConfiguration> element.

4. For each section you defined in step 2, add a new element to the <dbConfiguration> element.

For example, if you defined a section called `config1`, add a `config1` element. Each element has two attributes: dbParm and getIdentity. You can set either or both of these attributes.

The dbParm value sets the value of the DBParm parameter of the transaction object. It has a maximum length of 1000 characters. If you set a value for a parameter in the configuration file, any value that you set in code or in the Database Profile Setup dialog box is overridden.

The getIdentity value specifies the syntax used to retrieve the value of an identity column. It has a maximum length of 100 characters. If you do not specify a value for getIdentity, the `select @@identity` syntax is used.

Sample configuration file

This sample configuration file for PowerBuilder 17.0 is called *pb170.exe.config*. It contains three custom configurations. The <myconfig> element sets both the dbParm and getIdentity attributes. <myconfig1> sets getIdentity only, and <myconfig2> sets dbParm only. The <runtime> section is in the configuration file that ships with PowerBuilder but would not be included in the configuration file that you ship with your application, which would have the same name as your application with the extension *exe.config*.

```
<configuration>
   <configSections>
     <sectionGroup name="dbConfiguration">
        <section name="myconfig"
         type="Sybase.PowerBuilder.Db.DbConfiguration,
         Sybase.PowerBuilder.Db"
        />
        <section name="myconfig1"
         type="Sybase.PowerBuilder.Db.DbConfiguration,
         Sybase.PowerBuilder.Db"
        />
        <section name="myconfig2"
         type="Sybase.PowerBuilder.Db.DbConfiguration,
         Sybase.PowerBuilder.Db"
        />
      </sectionGroup>
   </configSections>

<runtime>
      <assemblyBinding xmlns=
       "urn:schemas-microsoft-com:asm.v1">
         <dependentAssembly>
            <assemblyIdentity name=
            "Sybase.PowerBuilder.Db"/>
            <codeBase href="file:///C:/Program Files/
             Appeon/PowerBuilder 17.0/DotNET/bin/
```

```
                Sybase.PowerBuilder.Db.dll"/>
            </dependentAssembly>
            <dependentAssembly>
              <assemblyIdentity name=
               "Sybase.PowerBuilder.WebService.WSDL"/>
              <codeBase href="file:///C:/Program Files/
               Appeon/PowerBuilder 17.0/DotNET/bin/
               Sybase.PowerBuilder.WebService.WSDL.dll"/>
            </dependentAssembly>
            <dependentAssembly>
              <assemblyIdentity name=
               "Sybase.PowerBuilder.WebService.Runtime"/>
               <codeBase href="file:///C:/Program Files/
                Appeon/PowerBuilder 17.0/DotNET/bin/
                Sybase.PowerBuilder.WebService.
                Runtime.dll"/>
            </dependentAssembly>
            <probing privatePath="DotNET/bin" />
         </assemblyBinding>
      </runtime>

      <dbConfiguration>
         <myconfig dbParm="disablebind=1"
          getIdentity="select scope_identity()"
         />
         <myconfig1 getIdentity="select scope_identity()"
         />
         <myconfig2 dbParm=
          "Namespace='Oracle.DataAccess.Client',
          DataSource='ora10gen',DisableBind=1,
          NCharBind=1,ADORelease='10.1.0.301'"
         />
      </dbConfiguration>
   </configuration>
```

Specifying the custom configuration to be used

On the System tab page in the Database Profile Setup dialog box for ADO.NET or in code, specify the name of the custom configuration section you want to use as the value of the DbConfigSection parameter. For example:

```
Sqlca.DBParm="DbConfigSection='myconfig'"
```

If you set any parameters in the profile or in code that are also set in the configuration file, the value specified in the configuration file takes precedence.

The configuration file must be present in the same directory as the executable file and must have the same name with the extension *.config*.

Sharing ADO.NET Database Connections

PowerBuilder applications can share database ADO.NET connections with third-party .NET assemblies exposed as COM through a connection proxy. The connection proxy is an instance of type IAdoConnectionProxy.

The IAdoConnectionProxy interface is defined in the *Sybase.PowerBuilder.DataSource.Sharing.dll* assembly as follows:

```
IAdoConnectionProxy {
 object Connection; //accepts System.Data.IDbConnection
 object Transaction; //accepts System.Data.IDbTransaction
 event EventHandler TransactionChanged;
}
```

Both the PowerBuilder application and the third-party assembly manage connections and transactions by referencing the proxy.

The assembly must be registered as COM by using *regasm.exe* under the Microsoft.NET\Framework\v4.0 folder. Please refer to the Microsoft MSDN library for information about *regasm.exe*.

The PowerBuilder Transaction object is the standard PowerBuilder nonvisual object used in database connections. To manage the shared connection, the Transaction object references the AdoConnectionProxy object using these methods:

- bool SetAdoConnection (oleobject *connectionProxy*) – accepts an imported ADO.NET connection.
- oleobject GetAdoConnection() – accepts an ADO.NET connection exported from the Transaction object.

Importing an ADO.NET Connection from a Third-Party .NET Assembly

You can import an ADO.NET connection from an external .NET assembly into a PowerBuilder application, enabling the application and the assembly to share the connection.

Use the SetAdoConnection method:

bool SetAdoConnection(oleobject *proxy*)

where *proxy* is the instance of type IAdoConnectionProxy that is passed in by the third-party assembly.

The imported connection and any transaction are assigned to the IAdoConnectionProxy instance.

The method returns true if the parameter is available (that is, the parameter is an instance of IAdoConnectionProxy or null). It returns false if the operation fails.

Start the connection after invoking SetAdoConnection.

Sample PowerScript Code

```
//Sample PowerScript code
SQLCA.DBMS = "ADO.NET"
SQLCA.AutoCommit = true
SQLCA.DBParm = "Namespace='System.Data.Odbc', DataSource='SQL
Anywhere 11 Demo'"
bool retVal = SQLCA.SetAdoConnection(emp.AdoConnectionProxy)
// emp is an instance of a type in the 3rd-party .NET assembly
if (retVal = true) then
  connect using SQLCA;
  // db operations
end if
```

Sample C# Code

Here is an example of C# code in the third-party assembly:

```
public class Emp {
  private IDbConnection conn;
  private IDbTransaction trans;
  ...
  private IAdoConnectionProxy proxy;
  ...
  public object AdoConnectionProxy {
    get {
      //disposing/clean-up actions.
      if (null == proxy) {
        proxy = new AdoConnectionProxy();
      }
      proxy.Connection = conn;
      proxy.Transaction = trans;
      return proxy;
    }
    set {
      //disposing/clean-up actions.
      proxy = value as IAdoConnectionProxy;
      if (null != proxy) {
        if (conn != proxy.Connection as IDbConnection)
        this.Disconnect();
        conn = proxy.Connection as IDbConnection;
        trans = proxy.Transaction as IDbTransaction;
        proxy.TransactionChanged += new
             EventHandler(proxy_TransactionChanged);
      } else {
        //disposing/clean-up actions.
      }
    }
  }
  ...
}
```

Exporting an ADO.NET Connection to a Third-Party .NET Assembly

To export an ADO.NET connection from a PowerBuilder application, use the GetAdoConnection method:

oleobject GetAdoConnection()

The method returns an instance of IAdoConnectionProxy. The proxy's ADO connection object is assigned to property IAdoConnectionProxy.Connection.

When a transaction starts, the proxy's active Transaction object is assigned to property IAdoConnectionProxy.Transaction, and AutoCommit is false. When AutoCommit is true, the exported IAdoConnectionProxy.Transaction is null..

The method returns null if the connection fails, and false if the operation fails.

To use the shared connection, your third-party assembly must reference the exported connection proxy and manage the transaction. To be notified when the active transaction is changed, you can subscribe the IAdoConnection.TransactionChanged event . Remember to close the connection.

Sample PowerScript Code

```
//Sample PowerScript code
SQLCA.DBMS = "ADO.NET"
SQLCA.AutoCommit = false
SQLCA.DBParm = "Namespace='System.Data.Odbc', DataSource='SQL
 Anywhere 11 Demo'"
Connect Using SQLCA;
emp.ConnectionProxy = SQLCA.GetAdoConnection()
// db operations
disconnect using SQLCA;
```

Sample C# Code

Here is an example of C# code in the third-party assembly:

```
// Manage the transaction
public class Emp {
  ...
  IAdoConnectionProxy proxy;
  IDbTransaction trans;
  ...
  public object ConnectionProxy {
    get { return proxy; }
    set {
      proxy = value as IAdoConnectionProxy;
      ...
      proxy.TransactionChanged += new
            EventHandler(proxy_TransactionChanged);
    }
  }
  void proxy_TransactionChanged(object sender, EventArgs e) {
    ...
    trans = sender as IDbTransaction;
    ...
  }
  ...
}
```

CHAPTER 6 Using the OData Interface

About this chapter

This chapter describes the OData interface and explains how to prepare to use this interface and how to define the OData database profile.

PowerBuilder can use OData datasources.

OData (the Open Data Protocol) is a Web protocol for querying and updating data that provides a way to unlock your data and free it from silos that exist in applications. OData is based on REST (Representational State Transfer). It uses the standard HTTP protocol to access data using GET, PUT, POST, and DELETE.

At runtime, users can retrieve and manipulate data.

Contents

Database Profile Setup - OData Dialog Box

Define a database profile to access an OData service in PowerBuilder using the OData interface.

Connection Tab

The Connection tab includes basic connection options that you must supply to access the information in the OData service.

- **Profile Name** The name of your database profile.
- **Connection Information** The uniform resource identifier (URI) that represents the OData service.

- **Authenticated Access** Select the type of access:
  - Anonymous access – Opens up the service so it can be accessed without providing credentials.
  - Integrated Windows authentication – Uses information on the client computer to validate the user's access.
  - Supply user ID and password – If you select this option, enter the user name required to connect to the datasource and the password for the specified login ID.
- **OData Extended Catalog** Use this option to define the extended attributes, such as edit styles and validation rules, that can be applied in columns of the tables. After you select this option, you can also specify the properties for each table and column. This information is saved in the registry.

Certificate Tab

The Certificate tab includes basic connection options that you must supply to access the information in the OData service.

Select one connection option:

- **No X509 Certificate** Select this option when the OData service does not require a certificate.
- **Select a certificate from current user's personal store** Click Change to select a certificate.
- **Specify a local certificate file** Click Browse to select a certificate file on the local machine.

Proxy Server Tab

The Proxy Server tab has additional connection options that you can use to manage access to the OData service.

- **Use proxy server settings in Tools \ Options dialog** Select this option when you need proxy or firewall settings to access the OData services. Before you select this option, set the proxy or firewall information in Tools > System Options > Firewall Setting. Be sure to select Use Above Values as System Defaults.

- **Bypass proxy server for local addresses** Use this option to bypass the proxy server when the OData Service is using a local address.

Preview Tab

The Preview tab provides a convenient way to generate correct PowerScript connection syntax in the PowerBuilder development environment for use in your PowerBuilder application script.

As you complete the Database Profile Setup dialog box for OData, the correct PowerScript connection syntax for each select option is built on the Preview tab. You can then copy the syntax you want from the Preview tab into your PowerBuilder application script.

- **Copy** Copies the selected text in the Database Connection Syntax box to the clipboard. You can then paste the syntax into your PowerBuilder script.

Database Painter

You can connect to the OData service and work with it in the Database painter.

The Database painter supports these features:

- Define an OData datasource with the database profile painter.
- Connect to an OData datasource in the Objects view.
- Add tables to the Object Layout view with drag-and-drop or the context menu.
- Display table and column properties in the Properties view with drag-and-drop or the context menu.
- Drag and drop a table onto the Columns view.
- Invoke the Edit Data feature in the Results view of the Database painter.

Since OData is not a SQL source, some features are not supported:

- The ISQL Session view is not enabled.
- You cannot create, modify, or delete tables.
- You cannot create users or groups of users.

- You cannot embed SQL statements in PowerScript.

Create a DataWindow Using an OData Service

❖ **To select the OData Service datasource in the DataWindow wizard:**

1. Select **File** > **New** from the menu bar and select DataWindow.
2. If there is more than one target, select the target where you want the DataWindow to be created from the drop-down list.
3. Choose the presentation style for the DataWindow object and click **Next**.
4. Select the **OData Service** datasource and click **Next**.
5. Select the OData profile and click **Next**.
   - You can select one table.
   - The Sort, Group, and Having tabs are not available.
   - The Results tab is obsolete, because it is used by PowerBuilder .NET.
   - In the Where tab you can specify some selection criteria using the WHERE clause for the SELECT statement.

   In the SQL painter.
6. Review your specifications and click **Finish**.

At runtime, the DataWindow or DataStore can manipulate OData service data, which includes retrieving, updating, inserting, and deleting the data.

Set the Connection Information for the OData Service

As with other databases, use the SQLCA Transaction object (or user-defined transaction object) to retrieve and display data from the OData service in a DataWindow or DataStore.

❖ **To retrieve and display data from the OData service in a DataWindow or DataStore:**

1. Set the appropriate values for the transaction object.

2. Connect to the OData service.
3. Set the transaction object for the DataWindow or DataStore.
4. Retrieve and update the data.
5. When the processes are complete, disconnect from the OData service.

The code looks something like this:

```
SQLCA.DBMS = "ODT"
SQLCA.DBParm = "ConnectString='URI=http://esx2-
appserver/TestDataService/Employee.svc'"
//connect to the service
connect using SQLCA;
dw_1.SetTransObject(SQLCA)
dw_1.Retrieve()
...
//disconnect from the service
disconnect using SQLCA;
```

For more information on using the global Transaction object, see Application Techniques.

PART 3

Working with Native Database Interfaces

This part describes how to set up and define database connections accessed through one of the native database interfaces.

CHAPTER 7 Using Native Database Interfaces

About this chapter

This chapter describes native database interfaces. The following chapters explain how to prepare to use the database and define any unique database interface parameters so that you can access your data.

Contents

About native database interfaces

The native database interfaces provide native connections to many databases and DBMSs. This chapter describes how the native database interfaces access these databases.

For a complete list of the supported native database interfaces, see "Database preferences and Supported Database Interfaces".

A native database interface is a direct connection to your data in PowerBuilder.

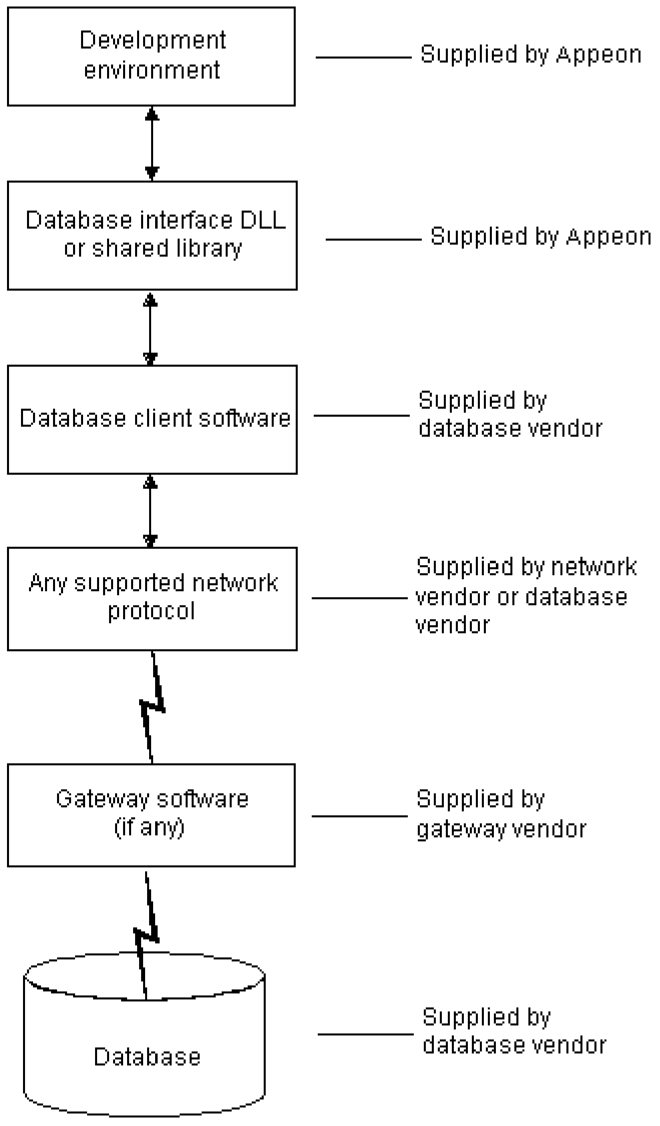
Each native database interface uses its own interface DLL to communicate with a specified database through a vendor-specific database API. For example, the SQL Native Client interface for Microsoft SQL Server uses a DLL named *PBSNC170.DLL* to access the database, whereas the Oracle 11*g* database interface accesses the database through *PBORA170.DLL*.

In contrast, a standard database interface uses a standard API to communicate with the database. For example, PowerBuilder can use a single-interface DLL (*PBODB170.DLL*) to communicate with the ODBC Driver Manager and corresponding driver to access any ODBC data source.

Components of a database interface connection

When you use a native database interface to access a database, your connection goes through several layers before reaching the data. Each layer is a separate component of the connection and each component might come from a different vendor.

Figure 7-1: Components of a database connection


Development environment
Supplied by Appeon
Database interface DLL or shared library
Supplied by Appeon
Database client software
Supplied by database vendor
Any supported network protocol
Supplied by network vendor or database vendor
Gateway software (if any)
Supplied by gateway vendor
Database
Supplied by database vendor

For diagrams showing the specific components of your connection, see "Basic software components" in the chapter for your native database interface.

Using a native database interface

You perform several basic steps to use a native database interface to access a database.

About preparing to use the database

The first step in connecting to a database through a native database interface is to prepare to use the database. Preparing the database ensures that you will be able to access and use your data in PowerBuilder.

You must prepare the database *outside* PowerBuilder *before* you start the product, then define the database interface and connect to it. The requirements differ for each database—but in general, preparing a database involves four basic steps.

❖ **To prepare to use your database with PowerBuilder:**

1. Make sure the required database server software is properly installed and configured at your site.
2. If network software is required, make sure it is properly installed and configured at your site and on the client computer so that you can connect to the database server.
3. Make sure the required database client software is properly installed and configured on the client computer. (Typically, the client computer is the one running PowerBuilder.)

   You must obtain the client software from your database vendor and make sure that the version you install supports *all* of the following:

   - The operating system running on the client computer
   - The version of the database that you want to access
   - The version of PowerBuilder that you are running
4. Verify that you can connect to the server and database you want to access outside PowerBuilder.

For specific instructions to use with your database, see "Preparing to use the database" in the chapter for your native database interface.

About installing native database interfaces

After you prepare to use the database, you must install the native database interface that accesses the database. See the instructions for each interface for more information.

About defining native database interfaces

Once you are ready to access the database, you start PowerBuilder and define the database interface. To define a database interface, you must create a database profile by completing the Database Profile Setup dialog box for that interface.

For general instructions, see About creating database profiles on page 7. For instructions about defining database interface parameters unique to a particular database, see “Preparing to use the database” in the chapter for your database interface.

For more information

The following chapters give general information about using each native database interface. For more detailed information:

- Check to see if there is a technical document that describes how to connect to your database.
- Ask your network or system administrator for assistance when installing and setting up the database server and client software at your site.

CHAPTER 8 Using Adaptive Server Enterprise

About this chapter

This section describes how to use the Adaptive Server Enterprise database interfaces in PowerBuilder.

Contents

Supported versions for Adaptive Server

You can access Adaptive Server versions 15.x, and 16.x using the SYC Adaptive Server database interface. Use of this interface to access other Open Server™ programs is not supported. The SYC database interface uses a DLL named *PBSYC170.DLL* to access the database through the Open Client™ CT-Lib API.

You can also access Adaptive Server version 15.x/16.x using the ASE Adaptive Server database interface. Use of this interface to access other Open Server programs is not supported. The Adaptive Server database interface uses a DLL named *PBASE170.DLL* to access the database through the Open Client CT-Lib API. To use this interface, the Adaptive Server 15/16 client must be installed on the client computer. The ASE interface supports large identifiers with up to 128 characters.

Client Library API
The Adaptive Server database interfaces use the Open Client CT-Library (CT-Lib) application programming interface (API) to access the database.

When you connect to an Adaptive Server database, PowerBuilder makes the required calls to the API. Therefore, you do not need to know anything about CT-Lib to use the database interface.

Supported Adaptive Server datatypes

The Adaptive Server interface supports the SAP datatypes listed in Table 8-1 in reports and embedded SQL.

Table 8-1: Supported datatypes for Adaptive Server Enterprise

Binary	NVarChar
BigInt (15.x and later)	Real
Bit	SmallDateTime
Char (see Column-length limits on page 83)	SmallInt
DateTime	SmallMoney
Decimal	Text
Double precision	Timestamp
Float	TinyInt
Identity	UniChar
Image	UniText (15.x and later)
Int	UniVarChar
Money	VarBinary
NChar	VarChar
Numeric	

In Adaptive Server 15.0 and later, PowerBuilder supports unsigned as well as signed bigint, int, and smallint datatypes. You can also use the following datatypes as identity columns in Adaptive Server 15.0 and later: bigint, int, numeric, smallint, tinyint, unsigned bigint, unsigned int, and unsigned smallint.

Accessing Unicode data

PowerBuilder can connect, save, and retrieve data in both ANSI/DBCS and Unicode databases. When character data or command text is sent to the database, PowerBuilder sends a DBCS string if the UTF8 database parameter is set to 0 (the default). If UTF8 is set to 1, PowerBuilder sends a UTF-8 string. The database server must be configured correctly to accept UTF-8 strings. See the description of the UTF8 database parameter in the online Help for more information.

The character set used by an Adaptive Server database server applies to all databases on that server. The nchar and nvarchar datatypes can store UTF-8 data if the server character set is UTF-8. The Unicode datatypes unichar and univarchar were introduced in Adaptive Server 12.5 to support Unicode data. Columns with these datatypes can store *only* Unicode data. Any data saved into such a column must be converted to Unicode explicitly. This conversion must be handled by the database server or client.

In Adaptive Server 12.5.1 and later, additional support for Unicode data has been added. For more information, see the documentation for your version of Adaptive Server.

Different display values in painters

The unichar and univarchar datatypes support UTF-16 encoding, therefore each unichar or univarchar character requires two bytes of storage. The following example creates a table with one unichar column holding 10 Unicode characters:

```
create table unitbl (unicol unichar(10))
```

In the Database painter, the column displays as `unichar(20)` because the column requires 20 bytes of storage. This is consistent with the way the column displays in SAP Central.

However, the mapping between the Type in the Column Specifications view in the Report painter and the column datatype of a table in the database is not one-to-one. The Type in the Column Specifications view shows the DataWindow® column datatype and DataWindow column length. The column length is the number of characters, therefore an Adaptive Server `unichar(20)` column displays as `char(10)` in the Column Specifications view.

Column-length limits

Adaptive Server 12.5 and earlier have a column-length limit of 255 bytes. Adaptive Server 12.5.x and later support wider columns for Char, VarChar, Binary, and VarBinary datatypes, depending on the logical page size and the locking scheme used by the server.

In PowerBuilder, you can use these wider columns for Char and VarChar datatypes with Adaptive Server 12.5.x when the following conditions apply:

- The Release database parameter is set to 12.5 or higher.

- You are accessing the database using Open Client 12.5.x or later.

The database must be configured to use a larger page size to take full advantage of the widest limits.

For more information about the Release database parameter, see "Release database parameter".

When you retrieve or update columns, PowerBuilder converts data appropriately between the Adaptive Server datatype and the PowerScript datatype. Similarly or identically named Adaptive Server and PowerScript datatypes do *not* necessarily have the same definitions. For information about the definitions of PowerScript datatypes, see the *PowerScript Reference*.

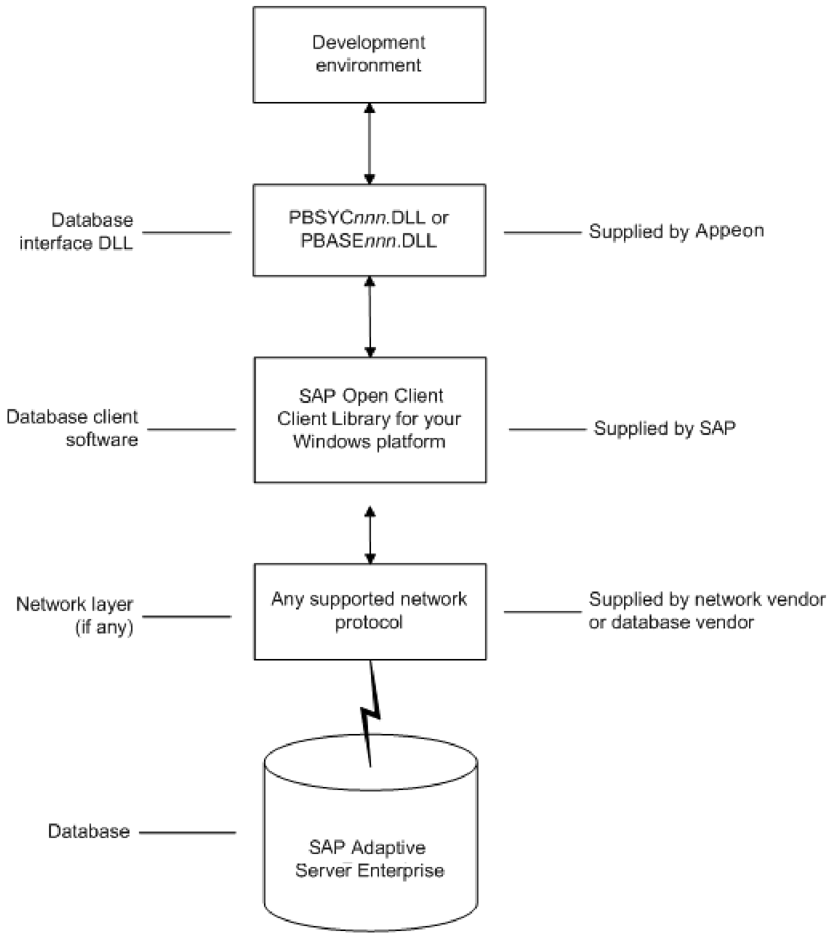
Conversion in PowerBuilder scripts

A double that has no fractional component is converted to a string with one decimal place if the converted string would cause Adaptive Server to have an overflow error when parsing the string. For example: the double value 12345678901234 would cause an overflow error, so PowerBuilder converts the double to the string value 12345678901234.0.

Basic software components for Adaptive Server

You must install the software components in Figure 8-1 to access an Adaptive Server database in PowerBuilder.

Figure 8-1: Components of an Adaptive Server Enterprise connection


Development environment
Database interface DLL
PBSYC*nnn*.DLL or PBASE*nnn*.DLL
Supplied by Appeon
Database client software
SAP Open Client Client Library for your Windows platform
Supplied by SAP
Network layer (if any)
Any supported network protocol
Supplied by network vendor or database vendor
Database
SAP Adaptive Server Enterprise

Preparing to use the Adaptive Server database

Before you define the interface and connect to an Adaptive Server database in PowerBuilder, follow these steps to prepare the database for use:

1. Install and configure the required database server, network, and client software.
2. Install the Adaptive Server database interface.
3. Verify that you can connect to Adaptive Server outside PowerBuilder.
4. Install the required PowerBuilder stored procedures in the sybsystemprocs database.

Preparing an Adaptive Server database for use with PowerBuilder involves these four basic tasks.

Step 1: Install and configure the database server

You must install and configure the database server, network, and client software for Adaptive Server.

❖ To install and configure the database server, network, and client software:

1. Make sure the Adaptive Server database software is installed on the server specified in your database profile.

   You must obtain the database server software from SAP.

   For installation instructions, see your Adaptive Server documentation.

2. Make sure the supported network software (for example, TCP/IP) is installed and running on your computer and is properly configured so that you can connect to the database server at your site.

   You must install the network communication driver that supports the network protocol and operating system platform you are using. The driver is installed as part of the Net-Library client software.

   For installation and configuration instructions, see your network or database administrator.

3 Install the required Open Client CT-Library (CT-Lib) software on each client computer on which PowerBuilder is installed.

You must obtain the Open Client software from SAP. Make sure the version of Open Client you install supports *all* of the following:

The operating system running on the client computer
The version of Adaptive Server that you want to access
The version of PowerBuilder that you are running

Required client software versions
To use the ASE Adaptive Server interface, you must install Open Client version 15.x or later. To use the SYC Adaptive Server interface, you must install Open Client version 11.x or later.

4 Make sure the Open Client software is properly configured so that you can connect to the database at your site.

Installing the Open Client software places the *SQL.INI* configuration file in the Adaptive Server directory on your computer.

SQL.INI provides information that Adaptive Server needs to find and connect to the database server at your site. You can enter and modify information in *SQL.INI* by using the configuration utility that comes with the Open Client software.

For information about setting up the *SQL.INI* or other required configuration file, see your Adaptive Server documentation.

5 If required by your operating system, make sure the directory containing the Open Client software is in your system path.

6 Make sure only one copy of each of the following files is installed on your client computer:

- Adaptive Server interface DLL
- Network communication DLL (for example, *NLWNSCK.DLL* for Windows Sockets-compliant TCP/IP)
- Database vendor DLL (for example, *LIBCT.DLL*)

Step 2: Install the database interface

In the PowerBuilder Setup program, select the Typical install, or select the Custom install and select the Adaptive Server Enterprise (ASE or SYC) database interface.

Step 3: Verify the connection

Make sure you can connect to the Adaptive Server database server and log in to the database you want to access from outside PowerBuilder.

Some possible ways to verify the connection are by running the following tools:

- **Accessing the database server** Tools such as the Open Client/Open Server Configuration utility (or any Ping utility) check whether you can reach the database server from your computer.
- **Accessing the database** Tools such as ISQL (interactive SQL utility) check whether you can log in to the database and perform database operations. It is a good idea to specify the same connection parameters you plan to use in your PowerBuilder database profile to access the database.

Step 4: Install the PowerBuilder stored procedures

PowerBuilder requires you to install certain stored procedures in the sybsystemprocs database *before* you connect to an Adaptive Server database for the first time. PowerBuilder uses these stored procedures to get information about tables and columns from the DBMS system catalog.

Run the SQL script or scripts required to install the PowerBuilder stored procedures in the sybsystemprocs database.

For instructions, see Installing stored procedures in Adaptive Server databases on page 95.

Defining the Adaptive Server database interface

To define a connection through the Adaptive Server interface, you must create a database profile by supplying values for at least the basic connection parameters in the Database Profile Setup - Adaptive Server Enterprise dialog box. You can then select this profile anytime to connect to your database in the development environment.

For information on how to define a database profile, see Using database profiles on page 6.

Using Open Client security services

The Adaptive Server interfaces provide several DBParm parameters that support Open Client 11.1.x or later network-based security services in your application. If you are using the required database, security, and PowerBuilder software, you can build applications that take advantage of Open Client security services.

What are Open Client security services?

Open Client 11.1.x or later **security services** allow you to use a supported third-party security mechanism (such as CyberSafe Kerberos) to provide login authentication and per-packet security for your application. Login authentication establishes a secure connection, and per-packet security protects the data you transmit across the network.

Requirements for using Open Client security services

For you to use Open Client security services in your application, *all of the following must be true*:

- You are accessing an Adaptive Server database server using Open Client Client-Library (CT-Lib) 11.1.x or later software.
- You have the required network security mechanism and driver.

  You have the required SAP-supported network security mechanism and SAP-supplied security driver properly installed and configured for your environment. Depending on your operating system platform, examples of supported security mechanisms include: Distributed Computing Environment (DCE) security servers and clients, CyberSafe Kerberos, and Windows NT LAN Manager Security Services Provider Interface (SSPI).

  For information about the third-party security mechanisms and operating system platforms that Appeon has tested with Open Client security services, see the Open Client documentation.
- You can access the secure server outside PowerBuilder.

  You must be able to access a secure Adaptive Server server using Open Client 11.1.x or later software from outside PowerBuilder.

To verify the connection, use a tool such as ISQL or SQL Advantage to make sure you can connect to the server and log in to the database with the same connection parameters and security options you plan to use in your PowerBuilder application.

- You are using aPowerBuilder database interface.

  You are using the ASE or SYC Adaptive Server interface to access the database.

- The Release DBParm parameter is set to the appropriate value for your database.

  You have set the Release DBParm parameter to 11or higher to specify that your application should use the appropriate version of the Open Client CT-Lib software.

  For instructions, see Release in the online Help.

- Your security mechanism and driver support the requested service.

  The security mechanism and driver you are using must support the service requested by the DBParm parameter.

Security services DBParm parameters

If you have met the requirements described in Requirements for using Open Client security services on page 89, you can set the security services DBParm parameters in the Database Profile Setup dialog box for your connection or in a PowerBuilder application script.

There are two types of DBParm parameters that you can set to support Open Client security services: login authentication and per-packet security.

Login authentication DBParms

The following login authentication DBParm parameters correspond to Open Client 11.1.x or later connection properties that allow an application to establish a secure connection.

Sec_Channel_Bind
Sec_Cred_Timeout
Sec_Delegation
Sec_Keytab_File
Sec_Mechanism
Sec_Mutual_Auth
Sec_Network_Auth
Sec_Server_Principal

Sec_Sess_Timeout

For instructions on setting these DBParm parameters, see their descriptions in the online Help.

Per-packet security DBParms

The following per-packet security DBParm parameters correspond to Open Client 11.1.x or later connection properties that protect each packet of data transmitted across a network. Using per-packet security services might create extra overhead for communications between the client and server.

Sec_Confidential
Sec_Data_Integrity
Sec_Data_Origin
Sec_Replay_Detection
Sec_Seq_Detection

For instructions on setting these DBParm parameters, see their descriptions in the online Help.

Using Open Client directory services

The Adaptive Server interfaces provide several DBParm parameters that support Open Client 11.1.x or later network-based directory services in your application. If you are using the required database, directory services, and PowerBuilder software, you can build applications that take advantage of Open Client directory services.

What are Open Client directory services?

Open Client 11.1.x or later **directory services** allow you to use a supported third-party directory services product (such as the Windows Registry) as your directory service provider. Directory services provide centralized control and administration of the network entities (such as users, servers, and printers) in your environment.

Requirements for using Open Client directory services

For you to use Open Client directory services in your application, *all of the following must be true*:

- You are accessing an Adaptive Server database server using Open Client Client-Library (CT-Lib) 11.x or later software
- You have the required SAP-supported directory service provider software and SAP-supplied directory driver properly installed and configured for your environment. Depending on your operating system platform, examples of supported security mechanisms include the Windows Registry and Distributed Computing Environment Cell Directory Services (DCE/CDS).

  For information about the directory service providers and operating system platforms that Appeon has tested with Open Client directory services, see the Open Client documentation.
- You must be able to access a secure Adaptive Server server using Open Client 11.1.x or later software from outside PowerBuilder.

  To verify the connection, use a tool such as ISQL or SQL Advantage to make sure you can connect to the server and log in to the database with the same connection parameters and directory service options you plan to use in your PowerBuilder application.
- You are using the ASE or SYC Adaptive Server interface to access the database.
- You must use the correct syntax as required by your directory service provider when specifying the server name in a database profile or PowerBuilder application script. Different providers require different syntax based on their format for specifying directory entry names.

  For information and examples for different directory service providers, see Specifying the server name with Open Client directory services next.
- You have set the Release DBParm to 11 or higher to specify that your application should use the behavior of the appropriate version of the Open Client CT-Lib software.

  For instructions, see Release database parameter in the online Help.
- The directory service provider and driver you are using must support the service requested by the DBParm.

Specifying the server name with Open Client directory services

When you are using Open Client directory services in a PowerBuilder application, you must use the syntax required by your directory service provider when specifying the server name in a database profile or PowerBuilder application script to access the database.

Different directory service providers require different syntax based on the format they use for specifying directory entry names. Directory entry names can be fully qualified or relative to the default (active) Directory Information Tree base (DIT base) specified in the Open Client/Server™ configuration utility.

The **DIT base** is the starting node for directory searches. Specifying a DITbase is analogous to setting a current working directory for UNIX or MS-DOS file systems. (You can specify a nondefault DIT base with the DS_DitBase DBParm. For information, see "DS_DitBase".)

Windows registry server name example

This example shows typical server name syntax if your directory service provider is the Windows registry.

```
Node name: SALES:software\sybase\server\SYS12
DIT base: SALES:software\sybase\server
Server name: SYS12
```

❖ To specify the server name in a database profile:

- Type the following in the Server box on the Connection tab in the Database Profile Setup dialog box. Do *not* start the server name with a backslash (\).

  ```
  SYS12
  ```

❖ To specify the server name in a PowerBuilder application script:

- Type the following. Do *not* start the server name with a backslash (\).

  ```
  SQLCA.ServerName = "SYS12"
  ```

  If you specify a value in the Server box in your database profile, this syntax displays on the Preview tab in the Database Profile Setup dialog box. You can copy the syntax from the Preview tab into your script.

DCE/CDS server name example

This example shows typical server name syntax if your directory service provider is Distributed Computing Environment Cell Directory Services (DCE/CDS).

```
Node name: /.../boston.sales/dataservers/sybase/SYS12
DIT base: /../boston.sales/dataservers
Server name: sybase/SYS12
```

❖ **To specify the server name in a database profile:**

- Type the following in the Server box on the Connection tab in the Database Profile Setup dialog box. Do *not* start the server name with a slash (/).

  ```
  sybase/SYS12
  ```

❖ **To specify the server name in a PowerBuilder application script:**

- Type the following. Do *not* start the server name with CN=.

  ```
  SQLCA.ServerName = "SYS12"
  ```

  If you specify a value in the Server box in your database profile, this syntax displays on the Preview tab in the Database Profile Setup dialog box. You can copy the syntax from the Preview tab into your script.

Directory services DBParm parameters

If you have met the requirements described in Requirements for using Open Client security services on page 89, you can set the directory services DBParms in a database profile for your connection or in a PowerBuilder application script.

The following DBParms correspond to Open Client 11.1.x or later directory services connection parameters:

DS_Alias
DS_Copy
DS_DitBase
DS_Failover
DS_Password (Open Client 12.5 or later)
DS_Principal
DS_Provider
DS_TimeLimit

For instructions on setting these DBParms, see their descriptions in the online Help.

Using PRINT statements in Adaptive Server stored procedures

The ASE or SYC Adaptive Server database interface allows you to use PRINT statements in your stored procedures for debugging purposes.

This means, for example, that if you turn on Database Trace when accessing the database through the ASE or SYC interface, PRINT messages appear in the trace log but they do not return errors or cancel the rest of the stored procedure.

Creating a report based on a cross-database join

The ability to create a report based on a heterogeneous cross-database join is available through the use of Adaptive Server's Component Integration Services. Component Integration Services allow you to connect to multiple remote heterogeneous database servers and define multiple proxy tables that reference the tables residing on those servers.

For information on how to create proxy tables, see the Adaptive Server documentation.

Installing stored procedures in Adaptive Server databases

This section describes how to install PowerBuilder stored procedures in an Adaptive Server Enterprise database by running SQL scripts provided for this purpose.

Appeon recommends that you run these scripts outside PowerBuilder *before* connecting to an Adaptive Server database for the first time through the Adaptive Server (ASE or SYC DBMS identifier) native database interface. Although the database interface will work without the PowerBuilder stored procedures created by these scripts, the stored procedures are required for full functionality.

What are the PowerBuilder stored procedure scripts?

What you do

In order to work with an Adaptive Server database in PowerBuilder, you or your system administrator should install certain stored procedures in the database *before* you connect to Adaptive Server from PowerBuilder *for the first time*.

You must run the PowerBuilder stored procedure scripts only once per database server, and not before each PowerBuilder session. If you have already installed the PowerBuilder stored procedures in your Adaptive Server database before connecting in PowerBuilder on any supported platform, you need *not* install the stored procedures again before connecting in PowerBuilder on a different platform.

PowerBuilder stored procedures

A **stored procedure** is a group of precompiled and preoptimized SQL statements that performs some database operation. Stored procedures reside on the database server where they can be accessed as needed.

PowerBuilder uses these stored procedures to get information about tables and columns from the Adaptive Server system catalog. (The PowerBuilder stored procedures are different from the stored procedures you might create in your database.)

SQL scripts

PowerBuilder provides SQL script files for installing the required stored procedures in the sybsystemprocs database:

Script	Use for
PBSYC.SQL	Adaptive Server databases
PBSYC2.SQL	Adaptive Server databases to restrict the Select Tables list

Where to find the scripts

The stored procedure scripts are located in the *Server* directory on the PowerBuilder installation package. The *Server* directory contains server-side installation components that are *not* installed with PowerBuilder on your computer.

PBSYC.SQL script

What it does

The *PBSYC.SQL* script contains SQL code that overwrites stored procedures that correspond to the same version of PowerBuilder in the Adaptive Server sybsystemprocs database and then re-creates them.

The *PBSYC.SQL* script uses the sybsystemprocs database to hold the PowerBuilder stored procedures. This database is created when you install Adaptive Server.

When to run it

Before you connect to an Adaptive Server database in PowerBuilder *for the first time* using the ASE or SYC DBMS identifier, you or your database administrator *must run* the *PBSYC.SQL* script once per database server into the sybsystemprocs database.

Run *PBSYC.SQL* if the server at your site will be accessed by anyone using the PowerBuilder or by deployment machines.

If you or your database administrator have already run the current version of *PBSYC.SQL* to install PowerBuilder stored procedures in the sybsystemprocs database on your server, you need not rerun the script to install the stored procedures again.

For instructions on running *PBSYC.SQL*, see How to run the scripts on page 99.

Stored procedures it creates

The *PBSYC.SQL* script creates the following PowerBuilder stored procedures in the Adaptive Server sybsystemprocs database. The procedures are listed in the order in which the script creates them.

PBSYC.SQL stored procedure	What it does
sp_pb170column	Lists the columns in a table.
sp_pb170pkcheck	Determines whether a table has a primary key.
sp_pb170fktable	Lists the tables that reference the current table.
sp_pb170procdesc	Retrieves a description of the argument list for a specified stored procedure.
sp_pb170proclist	Lists available stored procedures and extended stored procedures. If the SystemProcs DBParm parameter is set to 1 or Yes (the default), sp_pb170proclist displays both system stored procedures and user-defined stored procedures. If SystemProcs is set to 0 or No, sp_pb170proclist displays only user-defined stored procedures.
sp_pb170text	Retrieves the text of a stored procedure from the SYSCOMMENTS table.
sp_pb170table	Retrieves information about *all* tables in a database, including those for which the current user has no permissions. PBSYC.SQL contains the default version of sp_pb170table. If you want to replace the default version of sp_pb170table with a version that restricts the table list to those tables for which the user has SELECT permission, you can run the *PBSYC2.SQL* script, described in PBSYC2.SQL script next.
sp_pb170index	Retrieves information about all indexes for a specified table.

PBSYC2.SQL script

What it does

The *PBSYC2.SQL* script contains SQL code that drops and re-creates one PowerBuilder stored procedure in the Adaptive Server sybsystemprocs database: a replacement version of sp_pb170table.

The default version of sp_pb170table is installed by the *PBSYC.SQL* script. PowerBuilder uses the sp_pb170table procedure to build a list of *all* tables in the database, including those for which the current user has no permissions. This list displays in the Select Tables dialog box in PowerBuilder.

For security reasons, you or your database administrator might want to restrict the table list to display only those tables for which a user has permissions. To do this, you can run the *PBSYC2.SQL* script *after you run PBSYC.SQL*. *PBSYC2.SQL* replaces the default version of sp_pb170table with a new version that displays a restricted table list including only tables and views:

- Owned by the current user
- For which the current user has SELECT authority
- For which the current user's group has SELECT authority
- For which SELECT authority was granted to PUBLIC

When to run it

If you are accessing an Adaptive Server database using the ASE or SYC DBMS identifier in PowerBuilder, *you must first run PBSYC.SQL* once per database server to install the required PowerBuilder stored procedures in the sybsystemprocs database.

After you run *PBSYC.SQL*, you can optionally run *PBSYC2.SQL* if you want to replace sp_pb170table with a version that restricts the table list to those tables for which the user has SELECT permission.

If you do not want to restrict the table list, there is no need to run *PBSYC2.SQL*.

For instructions on running *PBSYC2.SQL*, see How to run the scripts on page 99.

Stored procedure it creates

The *PBSYC2.SQL* script creates the following PowerBuilder stored procedure in the Adaptive Server sybsystemprocs database:

PBSYC2.SQL stored procedure	What it does
sp_pb170table	Retrieves information about those tables in the database for which the current user has SELECT permission. This version of sp_pb170table replaces the default version of sp_pb170table installed by the *PBSYC.SQL* script.

How to run the scripts

You can use the ISQL or SQL Advantage tools to run the stored procedure scripts outside PowerBuilder.

Using ISQL to run the stored procedure scripts

ISQL is an interactive SQL utility that comes with the Open Client software on the Windows platforms. If you have ISQL installed, use the following procedure to run the PowerBuilder stored procedure scripts.

For complete instructions on using ISQL, see your Open Client documentation.

❖ **To use ISQL to run the PowerBuilder stored procedure scripts:**

1 Connect to the sybsystemprocs Adaptive Server database as the system administrator.

2 Open one of the following files containing the PowerBuilder stored procedure script you want to run:

PBSYC.SQL
PBSYC2.SQL

3 Issue the appropriate ISQL command to run the SQL script with the user ID, server name, and (optionally) password you specify. Make sure you specify uppercase and lowercase exactly as shown:

isql /U sa /S *SERVERNAME* **/i** *pathname* **/P** { *password* }

Parameter	Description
sa	The user ID for the system administrator. Do *not* change this user ID.
SERVERNAME	The name of the computer running the Adaptive Server database.
pathname	The drive and directory containing the SQL script you want to run.
password	(Optional) The password for the sa (system administrator) user ID. The default Adaptive Server installation creates the sa user ID without a password. If you changed the password for sa during the installation, replace *password* with your new password.

For example, if you are using PowerBuilder and are accessing the stored procedure scripts from `d:\server`, type either of the following:

```
isql /U sa /S TESTDB /i d:\server\pbsyb.sql /P
isql /U sa /S SALES /i d:\server\pbsyc.sql /P
```

```
adminpwd
```

Using SQL Advantage to run the stored procedure scripts

SQL Advantage is an interactive SQL utility that comes with the Open Client software on the Windows platform. If you have SQL Advantage installed, use the following procedure to run the PowerBuilder stored procedure scripts.

For complete instructions on using SQL Advantage, see your Open Client documentation.

❖ To use SQL Advantage to run the PowerBuilder stored procedure scripts:

1 Start the SQL Advantage utility.

2 Open a connection to the sybsystemprocs Adaptive Server database as the system administrator.

3 Open one of the following files containing the PowerBuilder stored procedure script you want to run:

PBSYC.SQL
PBSYC2.SQL

4 Delete the use sybsystemprocs command and the go command at the beginning of each script.

SQL Advantage requires that you issue the use sybsystemprocs command by itself, with no other SQL commands following it. When you open a connection to the sybsystemprocs database in step 2, you are in effect issuing the use sybsystemprocs command. This command should not be issued again as part of the stored procedure script.

Therefore, to successfully install the stored procedures, you *must* delete the lines shown in the following table from the beginning of the PowerBuilder stored procedure script *before* executing the script.

Before executing this script	Delete these lines

PBSYC.SQL	`use sybsystemprocs`
	`go`

PBSYC2.SQL	`use sybsystemprocs`
	`go`

5 Execute all of the statements in the SQL script.

6 Exit the SQL Advantage session.

CHAPTER 9 Using Informix

About this chapter

This chapter describes how to use the native IBM Informix database interfaces in PowerBuilder.

Contents

Supported versions for Informix

You can access the IBM Informix Dynamic Server (IDS) database version 10.x/12.x using the PowerBuilder I10 native Informix database interfaces. You can also access Informix OnLine and Informix Standard Engine (SE) databases.

The I10 interface in *PBI10170.DLL* requires the Informix Client SDK 2.9 or later for Informix application development and Informix Connect 2.9 or later for runtime deployment.

Supported Informix datatypes

The Informix database interfaces support the Informix datatypes listed in Table 9-1 in DataWindow objects and embedded SQL.

Table 9-1: Supported datatypes for Informix

Blob	LVarChar

Boolean	Money
Byte (a maximum of 2^31 bytes)	NChar
Char	NVarChar
Clob	Real
Date	Serial
DateTime	Serial8
Decimal	SmallInt (2 bytes)
Float	Text (a maximum of 2^31 bytes)
Int8	Time
Integer (4 bytes)	VarChar (1 to 255 bytes)
Interval	

Datatype conversion
When you retrieve or update columns, PowerBuilder converts data appropriately between the Informix datatype and the PowerScript datatype. Keep in mind, however, that similarly or identically named Informix and PowerScript datatypes do *not* necessarily have the same definitions.

For information about the definitions of PowerScript datatypes, see the *PowerScript Reference*.

Informix DateTime datatype

The DateTime datatype is a contiguous sequence of boxes. Each box represents a component of time that you want to record. The syntax is:

DATETIME *largest_qualifier* **TO** *smallest_qualifier*

PowerBuilder defaults to `Year TO Fraction(5)`.

For a list of qualifiers, see your Informix documentation.

❖ **To create your own variation of the DateTime datatype:**

1 In the Database painter, create a table with a DateTime column.

For instructions on creating a table, see the *Users Guide*.

2 In the Columns view, select Pending Syntax from the Objects or pop-up menu.

The Columns view displays the pending changes to the table definition. These changes execute only when you click the Save button to save the table definition.

3. Select Copy from the Edit or pop-up menu or click the Copy button.

   The SQL syntax (or the portion you selected) is copied to the clipboard.

4. In the ISQL view, modify the DateTime syntax and execute the CREATE TABLE statement.

   For instructions on using the ISQL view, see the *Users Guide*.

Informix Time datatype

The Informix database interfaces also support a time datatype. The time datatype is a subset of the DateTime datatype. The time datatype uses only the time qualifier boxes.

Informix Interval datatype

The interval datatype is one value or a sequence of values that represent a component of time. The syntax is:

INTERVAL *largest_qualifier* **TO** *smallest_qualifier*

PowerBuilder defaults to `Day(3) TO Day`. For more about interval datatypes, see your Informix documentation.

Features supported by the I10 interface

The interface of I10 supports several features that are not available when you use the IN9 interface. Some of these features require a specific version of the Informix Dynamic Server database.

Accessing Unicode data

PowerBuilder can connect, save, and retrieve data in ANSI/DBCS databases using the IN9 interface, but the IN9 interface does not support Unicode databases. The Informix I10 interface supports ANSI/DBCS and Unicode databases.

The I10 native interface uses the Informix GLS (Global Language Support) API for global language support. The native interface uses three DBParms to help you set up the locale used in the current connection:

- Client_Locale
- DB_Locale
- StrByCharset

These parameters are available on the Regional Settings tab page in the Database Profile Setup dialog box.

Client_Locale

Client_Locale specifies the value of the Informix environment variable CLIENT_LOCALE. The format is *language_territory.codeset*. For example:

```
Client_Locale='en_us.1252'
Client_Locale='en_us.utf8'
```

The I10 interface uses this setting to access string data in an Informix database and to process SQL statements. If you do not set the DBParm, the default locale value is based on the OS locale.

DB_Locale

DB_Locale specifies the value of the Informix environment variable DB_LOCALE. The format is *language_territory.codeset*. For example:

```
DB_Locale='en_us.1252'
DB_Locale='en_us.utf8'
```

DB_LOCALE specifies the language, territory, and code set that the database server needs to correctly interpret locale-sensitive datatypes such as NChar and NVarChar in a specific database. The code set specified in DB_LOCALE determines which characters are valid in any character column, as well as in the names of database objects such as databases, tables, columns, and views. If you do not set the DBParm, the I10 interface assumes that the DB_LOCALE value is the same as the CLIENT_LOCALE value.

You can set the CLIENT_LOCALE and DB_LOCALE environment variables directly using the Informix Setnet32 utility, available in the Utilities folder for the Informix database interfaces in the Objects view in the Database painter or the Database Profiles dialog box.

For more information about the Informix CLIENT_LOCALE and DB_LOCALE environment variables, see the *IBM Informix GLS User's Guide*, currently available at the Informix library Web site at http://publib.boulder.ibm.com/infocenter/idshelp/v111/index.jsp?topic=/com.ibm.glsug.doc/glsug.htm.

StrByCharset

The StrByCharset DBParm specifies how to convert string data between PowerBuilder Unicode strings and Informix client multibyte strings. By default, string conversion for UTF-8 code sets is based on the UTF-8 code set, and string conversion for non-UTF-8 code sets is based on the current OS code page. If StrByCharset is set to 1 (true), string conversion is based on the code set specified in the DBParm Client_Locale.

Assigning an owner to the PowerBuilder catalog tables

When you use the I10 interface, you can use the PBCatalogOwner DBParm on the System tab page to assign a nondefault owner to the extended attribute system tables. For ANSI-compliant databases, the owner name that you specify must be unique but the table name does not have to be unique. You can create multiple sets of catalog tables prefaced with different user names. However, if the database is not ANSI-compliant, the table name must be unique, so that only one set of catalog tables can be created with an assigned owner name.

Support for long object names

The I10 interface supports Informix long object names with up to 128 characters.

Renaming an index

With IDS 9.2.1 and later, you can change the name of an index in the Database painter when you are connected using the I10 interface. The I10 interface uses the IDS RENAME INDEX statement to change the name of the index. You need only drop and recreate the index if you want to make other changes.

SQL statement caching

In IDS 9.2.1 and later, the database server uses the SQL statement cache (SSC) to store SQL statements across user sessions. When any user executes a statement already stored in the SQL statement cache, the database server does not parse and optimize the statement again, resulting in improved performance. The statement must be a SELECT, UPDATE, DELETE, or INSERT statement, and it cannot contain user-defined routines.

There are several ways to configure caching on the server. The SET STATEMENT CACHE statement takes precedence over the STMT_CACHE environment variable and the STMT_CACHE configuration parameter. You must enable the SQL statement cache, either by setting the STMT_CACHE configuration parameter or by using the Informix onmode utility, *before* the SET STATEMENT CACHE statement can execute successfully.

You can set the StmtCache DBParm on the System tab page in the Database Profile Setup dialog box for I10 connections to turn SQL statement caching on or off on the client. However, the server must be configured to support SQL statement caching before you can access the cache from the client.

For more information about Informix SQL statement caching, see the IBM Informix Dynamic Server Performance Guide at http://publib.boulder.ibm.com/infocenter/idshelp/v111/index.jsp?topic=/com.ibm.glsug.doc/glsug.htm.

Creating and dropping indexes without locking

In IDS 10.0 and later, the SQL syntax of CREATE INDEX and DROP INDEX supports the ONLINE keyword to create or drop an index in an online environment where the database and its tables are continuously available. When you use the ONLINE keyword to create or drop an index, data definition language (DDL) operations execute without applying an exclusive lock on the table on which the specified index is defined.

If you use CREATE INDEX ONLINE to create an index on a table that other users are accessing, the index is not available until no users are updating the table.

If you issue DROP INDEX ONLINE to drop an index, no users can reference the index, but concurrent data manipulation language (DML) operations can use the index until the operations terminate. Dropping the index is deferred until no users are using the index.

You can set the OnlineIndex static DBParm on the System tab page in the Database Profile Setup dialog box for I10 connections to specify that the Database painter should use the ONLINE keyword when you create or drop an index.

Clustered index not supported
You cannot create a clustered index using online mode because it is not supported by IDS.

Column-level encryption

In IDS 10.0 and later, the SQL statement SET ENCRYPTION PASSWORD can improve the confidentiality of data and support data integrity by defining or resetting a password for encryption and decryption of data at the column level.

You can set the EncryptionPass and Hint static DBParms on the System tab page in the Database Profile Setup dialog box for I10 connections to specify a password and a hint to help you remember the password. The application uses built-in Informix functions to encrypt and decrypt character data.

Using multiple OUT parameters in user-defined routines

In a user-defined routine (UDR), an OUT parameter corresponds to a value returned through a pointer. Before IDS version 9.4, IDS supported no more than one OUT parameter in a UDR, and any OUT parameter was required to appear as the last item in the parameter list. IDS version 9.4 drops these restrictions, supporting multiple OUT parameters anywhere in the parameter list of the UDR. This feature is available when you use the I10 interface. It provides greater flexibility in defining UDRs, and removes the need to return collection variables in contexts where multiple returned values are required.

To return OUT parameters from a UDR, you must use statement local variables (SLVs).

In the following statement, the OUT parameter in the UDR myfunc is defined using the SLV syntax *slvname*#*out_param_type*.

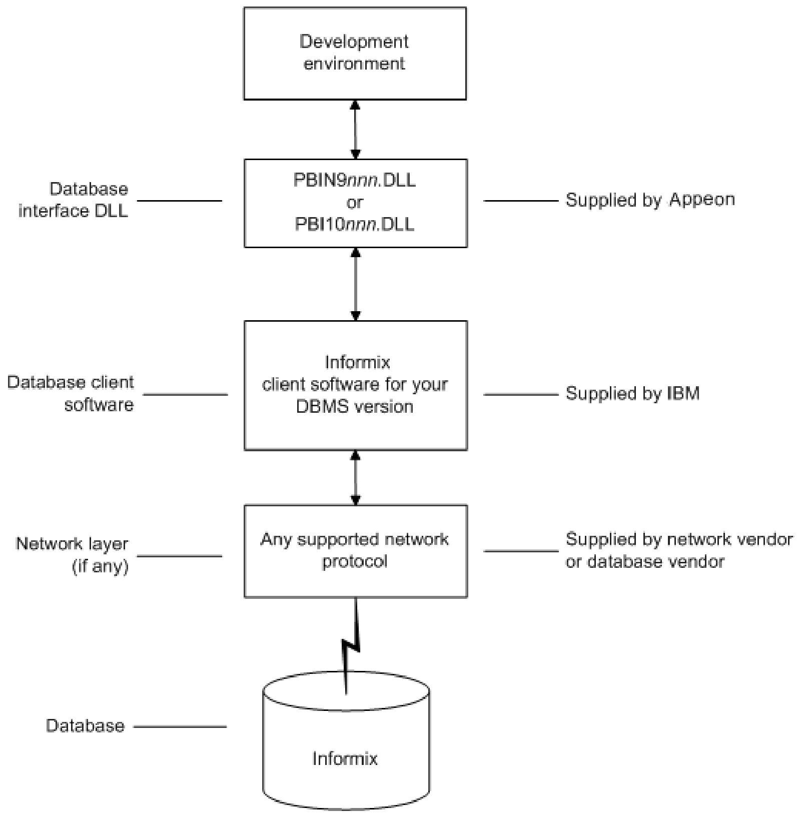
```
SELECT sales FROM mytable WHERE myfunc(10, sales#money)
< 1000
```

Informix does not support invoking a UDR with OUT parameters using an EXECUTE statement, therefore multiple OUT parameters are not supported in PowerBuilder remote procedure calls and embedded SQL EXECUTE PROCEDURE commands.

Basic software components for Informix

Figure 9-1 shows the basic software components required to access an Informix database using the native Informix database interfaces.

Figure 9-1: Components of an Informix connection


Development environment
Database interface DLL
PBIN9*nnn*.DLL or PBI10*nnn*.DLL
Supplied by Appeon
Database client software
Informix client software for your DBMS version
Supplied by IBM
Network layer (if any)
Any supported network protocol
Supplied by network vendor or database vendor
Database
Informix

Preparing to use the Informix database

Before you define the database interface and connect to an Informix database in PowerBuilder, follow these steps to prepare the database for use:

1 Install and configure the required database server, network, and client software.

2 Install the native Informix IN9 or I10 database interface.

3 Verify that you can connect to the Informix server and database outside PowerBuilder.

Step 1: Install and configure the database server

You must install and configure the required database server, network, and client software for Informix.

❖ To install and configure the required database server, network, and client software:

1 Make sure the Informix database server software and database network software is installed and running on the server specified in your database profile.

   You must obtain the database server and database network software from Informix.

   For installation instructions, see your Informix documentation.

2 Install the required Informix client software on each client computer on which PowerBuilder is installed.

   Install Informix Connect or the Informix Client SDK (which includes Informix Connect).

   You must obtain the Informix client software from IBM. Make sure the version of the client software you install supports *all* of the following:

   The operating system running on the client computer
   The version of the database that you want to access
   The version of PowerBuilder that you are running

   For installation instructions, see your Informix documentation.

3 Make sure the Informix client software is properly configured so that you can connect to the Informix database server at your site.

Run the SetNet32 utility to configure the client registry settings. When you configure Informix Connect client software, it creates a registry entry in *HKEY_LOCAL_MACHINE\Software\Informix\SqlHosts*. The registry entry contains parameters that define your network configuration, network protocol, and environment variables. If you omit these values from the database profile when you define the native Informix database interface, they default to the values specified in the registry entry.

For instructions on configuring your Informix client software, see your Informix documentation.

4 If required by your operating system, make sure the directory containing the Informix client software is in your system path.

Step 2: Install the database interface

In the PowerBuilder Setup program, select the Typical install, or select the native Informix database interface in the Custom install.

Step 3: Verify the connection

Make sure you can connect to the Informix server and database you want to access from outside PowerBuilder.

To verify the connection, use any Windows-based utility (such as the Informix ilogin.exe program) that connects to the database. When connecting, be sure to specify the same parameters you plan to use in your PowerBuilder database profile to access the database.

For instructions on using ilogin.exe, see your Informix documentation.

Defining the Informix database interface

To define a connection through an Informix database interface, you must create a database profile by supplying values for at least the basic connection parameters in the Database Profile Setup dialog box for Informix IN9 or I10. You can then select this profile at any time to connect to your database in the development environment.

For information on how to define a database profile, see Using database profiles on page 6.

Specifying the server name

When you specify the server name value, you *must* use the following format to connect to the database through the Informix interfaces:

host_name@*server_name*

Parameter	Description
host_name	The name of the host computer running the Informix database server. This corresponds to the Informix HOSTNAME environment variable.
server_name	The name of the server containing the Informix database. This corresponds to the Informix SERVER environment variable.

For example, to use a PowerBuilder native interface to connect to an Informix database server named server01 running on a host machine named `sales`, do either of the following:

- **In a database profile** Type the host name (`sales`) in the Host Name box and the server name (`server01`) in the Server box on the Connection tab in the Database Profile Setup dialog box. PowerBuilder saves this server name as `sales@server01` in the database profile entry in the system registry.
- **In a PowerBuilder script** Type the following in your PowerBuilder application script:

```
SQLCA.ServerName = "sales@server01"
```

Tip
If you specify a value for Host Name and Server in your database profile, this syntax displays on the Preview tab in the Database Profile Setup dialog box. You can then copy the syntax from the Preview tab into your script.

Accessing serial values in a PowerBuilder script

If you are connecting to an Informix database from a PowerBuilder script, you can obtain the serial number of the row inserted into an Informix table by checking the value of the SQLReturnData property of the Transaction object.

After an embedded SQL INSERT statement executes, SQLReturnData contains the serial number that uniquely identifies the row inserted into the table.

PowerBuilder updates SQLReturnData following an embedded SQL statement only; it does not update it following a DataWindow operation.

CHAPTER 10 Using Microsoft SQL Server

About this chapter

This chapter describes how to use the Microsoft SQL Server Native Client database interface in PowerBuilder.

Contents

Supported versions for SQL Server

You can access Microsoft SQL Server 2008 R2, 2012, 2014 or 2016 databases using the SQL Native Client interface. The SQL Native Client interface uses a DLL named *PBSNC170.DLL* to access the database. The interface uses the SQL Native Client (*sqlncli.h* and *sqlncli.dll*) on the client side and connects using OLE DB.

To take advantage of these features, you need to use the SNC interface. The SQL Server 2008 R2 or later SQL Native Client software must be installed on the client computer.

PBODB initialization file not used

Connections made directly through OLE DB use the PBODB initialization file to set some parameters, but connections made using the SNC interface do not depend on the PBODB initialization file.

Supported SQL Server datatypes

The SQL Native Client database interface supports the datatypes listed in Table 10-1.

Table 10-1: Supported datatypes for Microsoft SQL Server 2005

Binary	Real
Bit	SmallDateTime
Character (fewer than 255 characters)	SmallInt
DateTime	SmallMoney
Decimal	Text
Float	Timestamp
Identity	TinyInt
Image	VarBinary(*max*)
Int	VarBinary(*n*)
Money	VarChar(*max*)
Numeric	VarChar(*n*)
NVarChar(*max*)	XML
NVarChar(*n*)	

The XML datatype is a built-in datatype in SQL Server 2005 that enables you to store XML documents and fragments in a SQL Server database. The XML datatype maps to the PowerScript String datatype. You can use this datatype as a column type when you create a table, as a variable, parameter, or function return type, and with CAST and CONVERT functions.

Additional datatypes are supported for SQL Server 2008. For more information, see Support for new datatypes in SQL Server 2008 on page 123.

Datatype conversion

When you retrieve or update columns, PowerBuilder converts data appropriately between the Microsoft SQL Server datatype and the PowerScript datatype. Keep in mind, however, that similarly or identically named SQL Server and PowerScript datatypes do *not* necessarily have the same definitions.

For information about the definitions of PowerScript datatypes, see the *PowerScript Reference*.

In SQL Server 2005, the VarChar(*max*), NVarChar(*max*), and VarBinary(*max*) datatypes store very large values (up to 2^31 bytes). The VarChar(*max*) and NVarChar(*max*) datatypes map to the PowerScript String datatype and the VarBinary(*max*) datatype maps to the PowerScript Blob datatype. You can use these datatypes to obtain metadata, define new columns, and query data from the columns. You can also use them to pipeline data.

Working with large data values

For large data values of datatypes Text, NText, Image, Varchar(max), NVarchar(max), VarBinary(max), and XML, the SNC interface supports reading data directly from the database using an embedded SQL statement.

Example 1:

```
select image_col into :blob_var from mytable where
key_col = 1;
```

Example 2:

```
declare cur cursor for select id, image_col from
mytable;
open cur;
fetch cur into :id_var, :blob_var;
```

If the result set contains a large datatype of type Text or Varchar(max), using ANSI encoding, you must set a maximum size for each large value using the PBMaxBlobSize database parameter. For other large datatypes, there is no limitation on the size of the data. The SNC interface retrieves all the data from the database if there is sufficient memory.

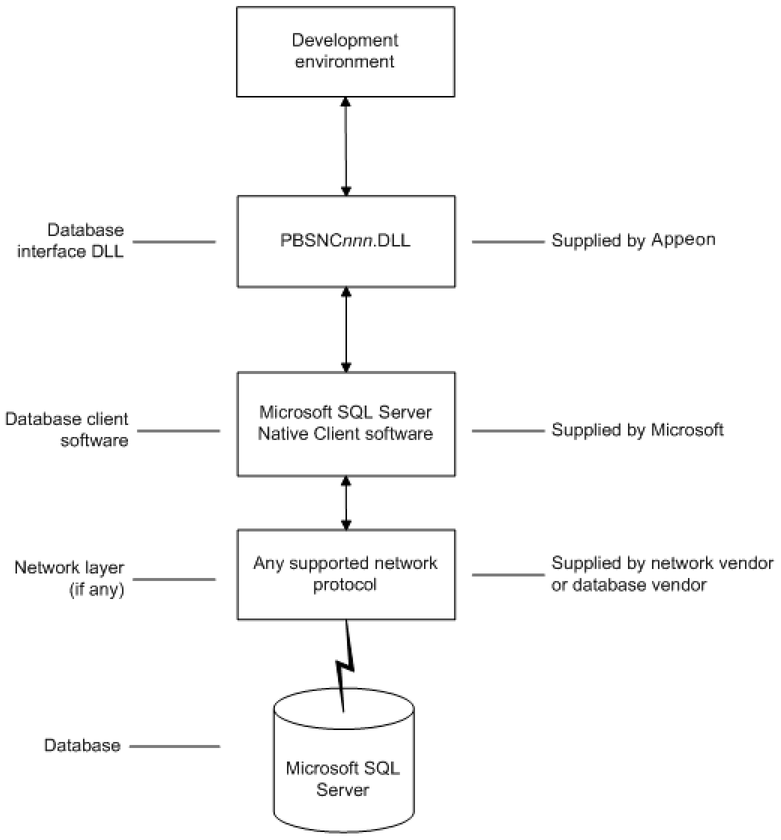
The SNC interface supports inserting and updating values of large datatypes using embedded SQL INSERT and UPDATE statements. You must set the DisableBind database parameter to 0 to enable the SNC interface to bind large data values. For example:

```
Insert into mytable (id, blob_col) values(1,
:blob_var);
Update mytable set blob_col = :blob_var where id = 1;
```

Basic software components for Microsoft SQL Server

You must install the software components in Figure 10-1 to access a database with the SQL Native Client interface. Microsoft SQL Server Native Client software contains a SQL OLE DB provider and ODBC driver in a single DLL.

Figure 10-1: Components of a Microsoft SQL Server connection



Preparing to use the SQL Server database

Before you define the database interface and connect to a Microsoft SQL Server database in PowerBuilder, follow these steps to prepare the database for use:

1 Install and configure the required database server, network, and client software.

2 Install the SQL Native Client database interface.

3 Verify that you can connect to the Microsoft SQL Server server and database outside PowerBuilder.

Step 1: Install and configure the database server

You must install and configure the database server, network, and client software for SQL Server.

❖ To install and configure the database server, network, and client software:

1 Make sure the Microsoft SQL Server database software is installed and running on the server specified in your database profile.

You must obtain the database server software and required licenses from Microsoft Corporation. For installation instructions, see your Microsoft SQL Server documentation.

Upgrading from an earlier version of SQL Server
For instructions on upgrading to a later version of SQL Server or installing it alongside an earlier version, see your Microsoft SQL Server documentation.

2 If you are accessing a remote SQL Server database, make sure the required network software (for example, TCP/IP) is installed and running on your computer and is properly configured so that you can connect to the SQL Server database server at your site.

For installation and configuration instructions, see your network or database administrator.

3 Install the required Microsoft SQL Native Client software on each client computer on which PowerBuilder is installed.

You must obtain the SQL Native Client software from Microsoft. Make sure the version of the client software you install supports *all* of the following:

The operating system running on the client computer
The version of the database that you want to access
The version of PowerBuilder that you are running

For installation instructions, see your Microsoft SQL Server documentation.

4 Make sure the SQL Native Client client software is properly configured so that you can connect to the SQL Server database server at your site.

For configuration instructions, see your Microsoft SQL Server documentation.

5 Make sure the directory containing the SQL Native Client software is in your system path.

6 Make sure only one copy of the *Sqlncli.dll* file is installed on your computer.

Step 2: Install the database interface

In the PowerBuilder Setup program, select the Custom install and select the SQL Native Client database interface.

Step 3: Verify the connection

Make sure you can connect to the SQL Server server and database you want to access from outside PowerBuilder.

To verify the connection, use any Windows-based utility that connects to the database. When connecting, be sure to specify the same parameters you plan to use in your PowerBuilder database profile to access the database.

Defining the SQL Server database interface

To define a connection through the SQL Native Client interface, you must create a database profile by supplying values for at least the basic connection parameters in the Database Profile Setup - SQL Native Client dialog box. You can then select this profile at any time to connect to your database in the development environment.

For information on how to define a database profile, see Creating a database profile on page 9. For new features that require special settings in the database profile, see SQL Server 2008 features on page 122. For a comparison of the database parameters you might have used with existing applications and those used with the SNC database interface, see Migrating from the MSS or OLE DB database interfaces next.

Migrating from the MSS or OLE DB database interfaces

In earlier releases of PowerBuilder, the MSS native interface was provided for connection to Microsoft SQL Server. This native interface was based on Microsoft DB-LIB functionality, which is no longer supported by Microsoft and is not Unicode-enabled. The MSS interface was removed in PowerBuilder 10.0.

Prior to the introduction of SQL Server 2005 and SQL Native Client, Microsoft recommended using the OLE DB database interface and MDAC to connect to SQL Server. You can continue to use this solution if you do not need to take advantage of new features in SQL Server 2005 or later versions.

This section provides a comparison between database parameters you might have used in existing applications with the parameters you can use with the SNC database interface.

MSS database parameters supported by SNC

Table 10-2 shows the database parameters and preferences that could be set in the Database Profile Setup dialog box for the discontinued MSS native database interface for Microsoft SQL Server, and indicates whether they are supported by the SNC interface.

The column on the left shows the tab page in the Database Profile Setup dialog box for MSS. The parameters and preferences may be on different tab pages in the SNC profile.

Table 10-2: MSS parameters supported by SNC

MSS	SNC
Connection tab:	
Language	Not supported
Lock	Supported (Transaction tab)
AutoCommit	Supported
CommitOnDisconnect	Supported
System tab:	

MSS	SNC
Log	Not supported
SystemProcs	Not supported
PBCatalogOwner	Supported
Transaction tab:	
Async	Not supported
DBGetTime	Not supported
CursorLock	Not supported
CursorScroll	Not supported
StaticBind	Supported
MaxConnect	Not supported
Syntax tab:	
DBTextLimit	Supported (as PBMaxTextSize on Transaction tab)
DateTimeAllowed	Not supported
OptSelectBlob	Not supported
Network tab:	
AppName	Supported (System tab)
Host	Supported (System tab)
PacketSize	Supported (System tab)
Secure	Supported (as TrustedConnection on General tab)

OLE DB database parameters supported by SNC

Table 10-3 shows the database parameters and preferences that can be set in the Database Profile Setup dialog box for the OLE DB standard interface for Microsoft SQL Server, and indicates whether they are supported by the SNC interface.

The column on the left shows the tab page in the Database Profile Setup dialog box for OLE DB. The parameters and preferences may be on different tab pages in the SNC profile.

Table 10-3: OLE DB parameters supported by SNC

OLE DB	SNC
Connection tab:	
Provider	Not supported
DataSource	Supported at runtime (as SQLCA.ServerName)
DataLink	Supported
Location	Not supported
ProviderString	Supported
System tab:	
PBCatalogOwner	Supported

OLE DB	SNC
ServiceComponents	Not supported
AutoCommit	Supported (General tab)
CommitOnDisconnect	Supported (General tab)
StaticBind	Supported (Transaction tab)
DisableBind	Supported (Transaction tab)
Init_Prompt	Not supported
TimeOut	Supported
LCID	Not supported
Transaction tab:	
Block	Supported
PBMaxBlobSize	Supported
Mode	Not supported
Lock	Supported
Syntax tab:	
DelimitIdentifier	Supported
IdentifierQuoteChar	Not supported
DateFormat	Supported
TimeFormat	Supported
DecimalSeparator	Supported
OJSyntax	Supported
Security tab:	
EncryptPassword	Not supported
CacheAuthentication	Not supported
PersistSensitive	Not supported
MaskPassword	Not supported
PersistEncrypted	Not supported
IntegratedSecurity	Supported (TrustedConnection on General tab)
ImpersonationLevel	Not supported
ProtectionLevel	Not supported

Additional database parameters

The SNC interface also supports the ReCheckRows and BinTxtBlob runtime-only parameters, the Encrypt, TrustServerCertificate, and SPCache parameters (on the System tab page), and the Identity parameter (on the Syntax tab page).

SPCache database parameter

You can control how many stored procedures are cached with parameter information by modifying the setting of the SPCache database parameter. The default is 100 procedures. To turn off caching of stored procedures, set SPCache to 0.

For more information about database parameters supported by the SNC interface, see *Connection Reference*.

SQL Server 2008 features

PowerBuilder support for connections to SQL Server 2008 databases includes new database parameters as well as support for new SQL Server datatypes. To connect to SQL Server 2008 from PowerBuilder, you must install the SNC 10.0 driver.

New database parameters

Provider parameter

The Provider DBParm parameter for the SQL Native Client (SNC) interface allows you to select the SQL Server version that you want to connect to. You can set this parameter in script to SQLNCLI (for the SNC 9.0 driver that connect to SQL Server 2005), to SQLNCLI10 (for the SNC 10.0 driver that connects to SQL Server 2008), or SQLNCLI11 (for the SNC 11.0 that connectc to SQL Server 2012 or later). Otherwise, you can select one of these providers on the Connection tab of the Database Profile Setup dialog box for the SNC interface.

If you do not set or select a provider, the default selection is SQLNCLI (SNC 9.0 for SQL Server 2005). This allows existing SNC interface users to be able to migrate to PowerBuilder 2017 without any modifications. If PowerBuilder fails to connect with the SQLNCLI provider, it will attempt to connect to SQLNCLI10 provider. However, if you explicitly set the provider and the connection fails, PowerBuilder displays an error message.

Failover parameter

The FailoverPartner DBParm parameter allows you to set the name of a mirror server, thereby maintaining database availability if a failover event occurs. You can also set the name of the mirror server on the System tab of the Database Profile Setup dialog box for the SNC interface.

When failover occurs, the existing PowerBuilder connection to SQL Server is lost. The SNC driver releases the existing connection and tries to reopen it. If reconnection succeeds, PowerBuilder triggers the failover event.

The following conditions must be satisfied for PowerBuilder to trigger the failover event:

- The FailoverPartner DBParm is supplied at connect time
- The SQL Server database is configured for mirroring
- PowerBuilder is able to reconnect successfully when the existing connection is lost

When failover occurs:

- PowerBuilder returns an error code (998) and triggers the failover event
- Existing cursors cannot be used and should be closed
- Any failed database operation can be tried again
- Any uncommitted transaction is lost. New transactions must be started

Support for new datatypes in SQL Server 2008

Date and time datatypes

The following table lists new SQL Server 2008 date and time datatypes and the PowerScript datatypes that they map to:

SQL Server datatype	PowerScript datatype
DATE	Date
TIME	Time (Supports only up to 6 fractional seconds precision although SQL Server datatype supports up to 7 fractional seconds precision.)
DATETIME2	DateTime (Supports only up to 6 fractional seconds precision although SQL Server datatype supports up to 7 fractional seconds precision.)

The SQL Server 2008 DATETIMEOFFSET datatype is not supported in PowerBuilder 2017.

Precision settings When you map to a table column in a SQL Server 2008 database, PowerBuilder includes a column labeled “Dec” in the Column view of the DataWindow painter, and a text box labeled “Fractional Seconds Precision” in the Column (Object Details) view of the Database painter. These fields allow you to list the precision that you want for the TIME and DATETIME2 columns.

The precision setting is for table creation only. When retrieving or updating the data in a column, PowerBuilder uses only up to six decimal places precision for fractional seconds, even if you enter a higher precision value for the column.

Filestream datatype

The FILESTREAM datatype allows large binary data to be stored directly in an NTFS file system. Transact-SQL statements can insert, update, query, search, and back up FILESTREAM data.

The SQL Server Database Engine implements FILESTREAM as a Varbinary(max) datatype. The PowerBuilder SNC interface maps the Varbinary(max) datatype to a BLOB datatype, so to retrieve or update filestream data, use the SelectBlob or UpdateBlob SQL statements, respectively. To specify that a column should store data on the file system, you must include the FILESTREAM attribute in the Varbinary(max) column definition. For example:

```
CREATE TABLE FSTest (
  GuidCol1 uniqueidentifier ROWGUIDCOL NOT NULL
  UNIQUE DEFAULT NEWID(),
  IntCol2 int,
  varbinaryCol3 varbinary(max) FILESTREAM);
```

Do not use PowerScript file access functions with FILESTREAM data
You can access FILESTREAM data by declaring and using the Win32 API functions directly in PowerBuilder applications. However, existing PowerBuilder file access functions cannot be used to access FILESTREAM files. For more information about accessing FILESTREAM data using Win32 APIs, see the MSDN SQL Server Developer Center Web site at http://msdn.microsoft.com/en-us/library/bb933877(SQL.100).aspx.

Using CLR datatypes in PowerBuilder

The binary values of the .NET Common Language Runtime (CLR) datatypes can be retrieved from a SQL Server database as blobs that you could use in PowerBuilder applications to update other columns in the database. If their return values are compatible with PowerBuilder datatypes, you can use CLR datatype methods in PowerScript, dynamic SQL, embedded SQL or in DataWindow objects, because the SQL script is executed on the SQL Server side.

The CLR datatypes can also be mapped to Strings in PowerScript, but the retrieved data is a hexadecimal string representation of binary data.

You can use the ToString method to work with all datatypes that are implemented as CLR datatypes, such as the HierarchyID datatype, Spatial datatypes, and User-defined types.

HierarchyID datatype

HierarchyID is a variable length, system datatype that can store values representing nodes in a hierarchical tree, such as an organizational structure. A value of this datatype represents a position in the tree hierarchy.

ISQL Usage You can use HierarchyID columns with CREATE TABLE, SELECT, UPDATE, INSERT, and DELETE statements in the ISQL painter. For example:

```
CREATE TABLE Emp (
  EmpId int NOT NULL,
  EmpName varchar(20) NOT NULL,
  EmpNode hierarchyid NULL);
```

To insert HierarchyID data, you can use the canonical string representation of HierarchyID or any of the methods associated with the HierarchyID datatype as shown below.

```
INSERT into Emp VALUES (1, 'Scott',
  hierarchyid::GetRoot());
INSERT into Emp VALUES (2, 'Tom' , '/1/');

DECLARE @Manager hierarchyid
SELECT @Manager = hierarchyid::GetRoot() FROM Emp
INSERT into Emp VALUES (2, 'Tom',
  @Manager.GetDescendant(NULL,NULL));
DECLARE @Employee hierarchyid
SELECT @Employee = CAST('/1/2/3/4/' AS hierarchyid)
INSERT into Emp VALUES (2, 'Jim' , @Employee);
```

You cannot select the HierarchyID column directly since it has binary data, and the ISQL painter Results view does not display binary columns. However, you can retrieve the HierarchyID data as a string value using the ToString method of HierarchyID. For example:

```
Select EmpId, EmpName, EmpNode.ToString() from Emp;
```

You can also use the following methods on HierarchyID columns to retrieve its data: GetAncestor, GetDescendant, GetLevel, GetRoot, IsDescendant, Parse, and Reparent. If one of these methods returns a HierarchyID node, then use ToString to convert the data to a string. For example:

```
Select EmpId, EmpName, EmpNode.GetLevel() from Emp;
Select EmpId, EmpName,
   EmpNode.GetAncestor(1).ToString() from Emp;
```

HierarchyID columns can be updated using a String value or a HierarchyID variable:

```
Update Emp Set EmpNode = '/1/2/' where EmpId=4;
```

```
Delete from Emp where EmpNode = '/1/2/';
```

PowerScript Usage You can use HierarchyID columns in embedded SQL statements for SELECT, INSERT, UPDATE, and DELETE operations. HierarchyID data can be retrieved either as a String or as a Binary(Blob) datatype using the SelectBlob statement.

When using a String datatype to retrieve HierarchyID data, use the ToString method. Otherwise the data will be a hexadecimal representation of the binary HierarchyID value.

The following example shows how you can use HierarchyID methods in embedded SQL:

```
long id
String hid,name
Select EmpId, EmpName, EmpNode.ToString()
  into :id, :name, :hid
  from Emp where EmpId=3;
Select EmpId, EmpName, EmpNode.GetLevel()
  into :id, :name, :hid
  from Emp where EmpId=3;
Blob b
Selectblob EmpNode into :b from Emp where EmpId =2;
```

DataWindow Usage DataWindow objects do not directly support the HierarchyID datatype. But you can convert the HierarchyID to a string using the ToString method or an associated HierarchyID method in the data source SQL. For example:

```
SELECT EmpId, EmpName, EmpNode.ToString() FROM Emp;
SELECT EmpId, EmpName, EmpNode.GetLevel() FROM Emp;
```

Spatial datatypes

Microsoft SQL Server 2008 supports two spatial datatypes: the geometry datatype and the geography datatype. In SQL Server, these datatypes are implemented as .NET Common Language Runtime (CLR) datatypes.

Although the PowerBuilder SNC interface does not work with CLR datatypes, you can convert the spatial datatypes into strings (with the ToString function) and use them in PowerScript, in the ISQL painter, in embedded SQL, and in DataWindow objects. This is similar to the way you use the HierarchyID datatype. The SelectBlob SQL statement also lets you retrieve binary values for these datatypes.

The geography and geometry datatypes support eleven different data objects, or instance types, but only seven of these types are instantiable: Points, LineStrings, Polygons, and the objects in an instantiable GeometryCollection (MultiPoints, MultiLineStrings, and MultiPolygons). You can create and work with these objects in a database, calling methods associated with them, such as STAsText, STArea, STGeometryType, and so on.

For example:

```
CREATE TABLE SpatialTable (id int IDENTITY (1,1),
  GeomCol geometry);
INSERT INTO SpatialTable (GeomCol) VALUES (
  geometry::STGeomFromText(
   'LINESTRING (100 100,20 180,180 180)',0));
select id, GeomCol.ToString() from SpatialTable;
select id, GeomCol.STAsText(),
  GeomCol.STGeometryType(),
  GeomCol.STArea() from SpatialTable;
```

User-defined types

User-defined types (UDTs) are implemented in SQL Server as CLR types and integrated with .NET. Microsoft SQL Server 2008 eliminates the 8 KB limit for UDTs, enabling the size of UDT data to expand dramatically.

Although the PowerBuilder SNC interface does not directly support UDT datatypes, you can use the ToString method to retrieve data for UDTs in the same way as for other CLR datatypes such as HierarchyId or the spatial datatypes. However, if a UDT datatype is mapped to a String datatype in PowerScript, UDT binary values will be retrieved as hexadecimal strings. To retrieve or update data in binary form (blob) from a UDT, you can use the SelectBlob or UpdateBlob SQL statements, respectively.

You can use any of the associated methods of UDT or CLR datatypes that return compatible data (such as String, Long, Decimal, and so on) for PowerBuilder applications.

T-SQL enhancements

MERGE statement

The MERGE Transact-SQL statement performs INSERT, UPDATE, or DELETE operations on a target table or view based on the results of a join with a source table. You can use MERGE statement in the ISQL painter and in PowerScript using dynamic SQL. For example

```
String mySQL
mySQL = "MERGE INTO a USING b ON a.keycol = b.keycol " &
  + "WHEN MATCHED THEN "&
  + "UPDATE SET col1 = b.col1,col2 = b.col2 " &
  + "WHEN NOT MATCHED THEN " &
  + "INSERT (keycol, col1, col2, col3)" &
  + "VALUES (b.keycol, b.col1, b.col2, b.col3) " &
  + "WHEN SOURCE NOT MATCHED THEN " &
  + "DELETE;"
EXECUTE IMMEDIATE :Mysql;
```

Using the MERGE statement in ISQL
A MERGE statement must be terminated by a semicolon. By default the ISQL painter uses a semicolon as a SQL terminating character, so to use a MERGE statement in ISQL, the terminating character must be changed to a colon (:), a forward slash (/), or some other special character.

Grouping sets

GROUPING SETS is an extension of the GROUP BY clause that lets you define multiple groupings in the same query. GROUPING SETS produce a single result set, making aggregate querying and reporting easier and faster. It is equivalent to a UNION ALL operation for differently grouped rows.

The GROUPING SETS, ROLLUP, and CUBE operators are added to the GROUP BY clause. A new function, GROUPING_ID, returns more grouping-level information than the existing GROUPING function. (The WITH ROLLUP, WITH CUBE, and ALL syntax is not ISO compliant and is therefore deprecated.)

The following example uses the GROUPING SETS operator and the GROUPING_ID function:

```
SELECT EmpId, Month, Yr, SUM(Sales) AS Sales
  FROM Sales
  GROUP BY GROUPING SETS((EmpId, ROLLUP(Yr, Month)));
SELECT COL1, COL2,
  SUM(COL3) AS TOTAL_VAL,
  GROUPING(COL1) AS C1,
  GROUPING(COL2) AS C2,
  GROUPING_ID(COL1, COL2) AS GRP_ID_VALUE
  FROM TEST_TBL GROUP BY ROLLUP (COL1, COL2);
```

You can use the GROUPING SETS operator in the ISQL painter, in PowerScript (embedded SQL and dynamic SQL) and in DataWindow objects (syntax mode).

Row constructors

Transact-SQL now allows multiple value inserts within a single INSERT statement. You can use the enhanced INSERT statement in the ISQL painter and in PowerScript (embedded SQL and dynamic SQL). For example:

```
INSERT INTO Employees VALUES ('tom', 25, 5),
  ('jerry', 30, 6), ('bok', 25, 3);
```

When including multiple values in a single INSERT statement with host variables, you must set the DisableBind DBParm to 1. If you use literal values as in the above example, you can insert multiple rows in a single INSERT statement regardless of the binding setting.

Compatibility level

In SQL Server 2008, the ALTER DATABASE statement allows you to set the database compatibility level (SQL Server version), replacing the sp_dbcmptlevel procedure. You can use this syntax in the ISQL painter and in PowerScript (dynamic SQL). For example:

```
ALTER DATABASE <database_name>
  SET COMPATIBILITY_LEVEL = {80 | 90 | 100}
80 = SQL Server 2000
90 = SQL Server 2005
100 = SQL Server 2008
```

Compatibility level affects behaviors for the specified database only, not for the entire database server. It provides only partial backward compatibility with earlier versions of SQL Server. You can use the database compatibility level as an interim migration aid to work around differences in the behaviors of different versions of the database.

Table hints

The FORCESEEK table hint overrides the default behavior of the query optimizer. It provides advanced performance tuning options, instructing the query optimizer to use an index seek operation as the only access path to the data in the table or view that is referenced by the query. You can use the FORCESEEK table hint in the ISQL painter, in PowerScript (embedded SQL and dynamic SQL), and in DataWindow objects (syntax mode).

For example:

```
Select ProductID, OrderQty from SalesOrderDetail
   with (FORCESEEK);
```

Unsupported SQL Server 2008 features

The PowerBuilder SNC interface does not support the User-Defined Table Type (a user-defined type that represents the definition of a table structure) that was introduced in SQL Server 2008.

Notes on using the SNC interface

Using the DBHandle PowerScript function

The DBHandle function on the Transaction object returns the IUnknown* interface of the current session object. You can use this interface to query any interface in the session object. The interface is not locked by pIUnknown->Addref() in PowerBuilder, therefore you should not call the pIUnknown->Release() to free the interface after using it.

SQL batch statements

The SNC interface supports SQL batch statements. However, they must be enclosed in a BEGIN...END block or start with the keyword DECLARE:

- Enclosed in a BEGIN...END block:

```
BEGIN
INSERT INTO t_1 values(1, 'sfdfs')
INSERT INTO t_2 values(1, 'sfdfs')
SELECT * FROM t_1
SELECT * FROM t_2
END
```

- Starting with the keyword DECLARE:

```
DECLARE @p1 int, @p2 varchar(50)
SELECT  @p1 = 1
EXECUTE  sp_4 @p1, @p2 OUTPUT
SELECT @p2  AS  'output'
```

You can run the batch of SQL statements in the Database painter or in PowerScript. For example:

```
String batchSQL //contains a batch of SQL statements
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
PREPARE SQLSA FROM :batchSQL ;
OPEN DYNAMIC my_cursor ;
//first result set
FETCH my_cursor INTO . . .
//second result set
FETCH my_cursor INTO . .
. . .
CLOSE my_cursor ;
```

Connection pooling

The SNC interface pools connections automatically using OLE DB pooling. To disable OLE DB pooling, type the following in the Extended Properties box on the Connection tab page in the Database Profile Setup dialog box:

```
OLE DB Services=-4
```

You can also type the following statement in code:

```
ProviderString='OLE DB Services=-4')
```

Triggers and synonyms in the Database painter

In the Objects view for SNC profiles in the Database painter, triggers display for tables in the Tables folder and Microsoft SQL Server 2005 synonyms display for tables and views.

CHAPTER 11 Using Oracle

About this chapter

This chapter describes how to use the native Oracle database interfaces in PowerBuilder.

Contents

Supported versions for Oracle

PowerBuilder provides three Oracle database interfaces. These interfaces use different DLLs and access different versions of Oracle.

Table 11-1: Supported native database interfaces for Oracle

Oracle interface	DLL
O10 Oracle 10*g*	*PBO10170.DLL*
ORA Oracle 11*g*/12*c*	*PBORA170.DLL*

The ORA database interface allows you to connect to Oracle 11*g*/12*c* servers using Oracle 11*g*/12*c* Database Client or Oracle 11*g*/12*c* Instant Client. It includes partial support for the XMLType datatype that it maps to the PowerBuilder String datatype. It also supports session and connection pooling, load balancing, the Oracle Client Cache, setting of an application driver name, and access through a proxy. Oracle 11*g* clients can also connect to Oracle 10*g* servers.

The O10 database interface allows you to connect to Oracle 10*g* servers using Oracle 10*g* Database Client or Oracle 10*g* Instant Client. It supports BINARY_FLOAT and BINARY_DOUBLE datatypes and increased size limits for CLOB and NCLOB datatypes. Oracle 10*g* clients can connect to Oracle 10*g* servers.

Supported Oracle datatypes

The Oracle database interfaces support the Oracle datatypes listed in Table 11-2 in reports and embedded SQL.

Table 11-2: Supported datatypes for Oracle

Binary_Float (Oracle 10*g* and later only)	LongRaw
Binary_Double (Oracle 10*g* and later only)	NChar
Bfile	Number
Blob	NVarChar2
Char	Raw
Clob	TimeStamp
Date	VarChar
Float	VarChar2
Long	XMLType (partial support, ORA driver only)

The ORA driver adds support for the XMLType datatype that was introduced with Oracle 9*i*. However, you cannot use this datatype with embedded SQL statements or in a report.

Accessing Unicode data

PowerBuilder can connect, save, and retrieve data in both ANSI/DBCS and Unicode databases, but it does not convert data between Unicode and ANSI/DBCS. When character data or command text is sent to the database, PowerBuilder sends a Unicode string. The driver must guarantee that the data is saved as Unicode data correctly. When PowerBuilder retrieves character data, it assumes the data is Unicode.

A Unicode database is a database whose character set is set to a Unicode format, such as UTF-8, UTF-16, UCS-2, or UCS-4. All data must be in Unicode format, and any data saved to the database must be converted to Unicode data implicitly or explicitly.

A database that uses ANSI (or DBCS) as its character set might use special datatypes to store Unicode data. These datatypes are NCHAR and NVARCHAR2. Columns with this datatype can store *only* Unicode data. Any data saved into such a column must be converted to Unicode explicitly. This conversion must be handled by the database server or client.

A constant string is regarded as a char type by Oracle and its character set is NLS_CHARACTERSET. However, if the datatype in the database is NCHAR and its character set is NLS_NCHAR_CHARACTERSET, Oracle performs a conversion from NLS_CHARACTERSET to NLS_NCHAR_CHARACTERSET. This can cause loss of data. For example, if NLS_CHARACTERSET is WE8ISO8859P1 and NLS_NCHAR_CHARACTERSET is UTF8, when the Unicode data is mapped to WE8ISO8859P1, the Unicode data is corrupted.

If you want to access Unicode data using NCHAR and NVARCHAR2 columns or stored procedure parameters, use PowerBuilder variables to store the Unicode data in a script using embedded SQL to avoid using a constant string, and force PowerBuilder to bind the variables.

By default, the Oracle database interfaces bind all string data to internal variables as the Oracle CHAR datatype to avoid downgrading performance. To ensure that NCHAR and NVARCHAR2 columns are handled as such on the server, set the NCharBind database parameter to 1 to have the drivers bind string data as the Oracle NCHAR datatype.

For example, suppose table1 has a column c1 with the datatype NVARCHAR2. To insert Unicode data into the table, set DisableBind to 0, set NCharBind to 1, and use this syntax:

```
string var1
insert into table1 (c1) values(:var1);
```

If an Oracle stored procedure has an NCHAR or NVARCHAR2 input parameter and the input data is a Unicode string, set the BindSPInput database parameter to 1 to force the Oracle database to bind the input data. The Oracle database interfaces are able to describe the procedure to determine its parameters, therefore you do not need to set the NCharBind database parameter.

For a report to access NCHAR and NVARCHAR2 columns and retrieve data correctly, set both DisableBind and StaticBind to 0. Setting StaticBind to 0 ensures that PowerBuilder gets an accurate datatype before retrieving.

TimeStamp datatype

The TimeStamp datatype in Oracle9*i* and later is an extension of the Date datatype. It stores the year, month, and day of the Date value plus hours, minutes, and seconds:

Timestamp[*fractional_seconds_precision*]

The *fractional_seconds_precision* value is optional and provides the number of digits for indicating seconds. The range of valid values for use with PowerBuilder is 0-6.

Datatype conversion

When you retrieve or update columns, in general PowerBuilder converts data appropriately between the Oracle datatype and the PowerScript datatype. Keep in mind, however, that similarly or identically named Oracle and PowerScript datatypes do *not* necessarily have the same definitions.

For information about the definitions of PowerScript datatypes, see the *PowerScript Reference*.

Number datatype converted to decimal

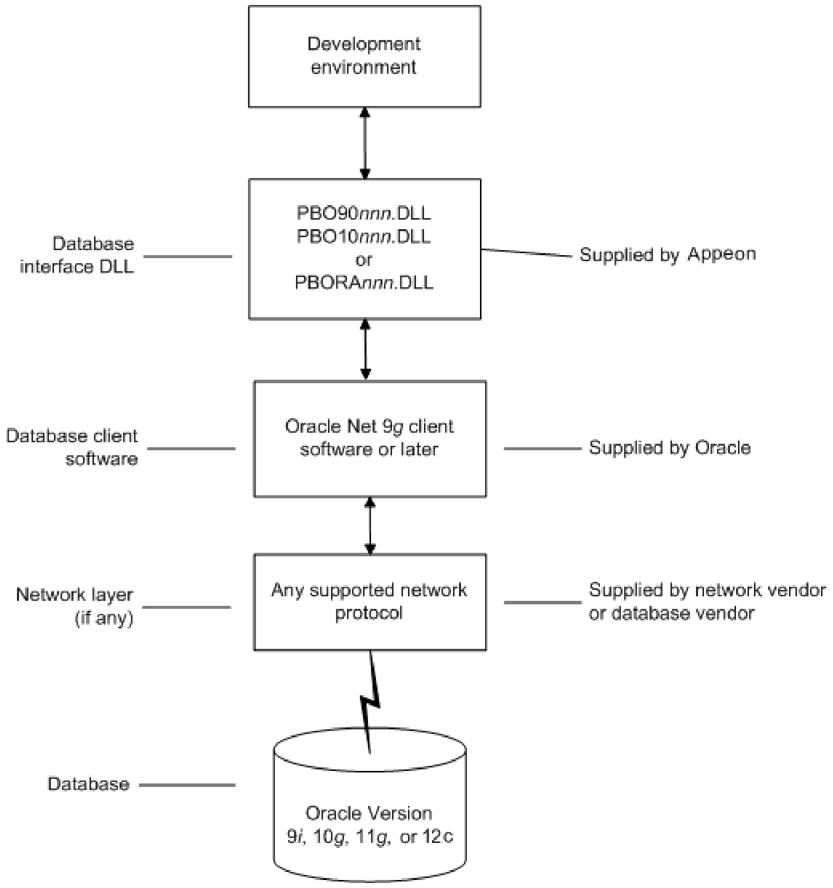
When a DataWindow object is defined in PowerBuilder, the Oracle datatype number(size,d) is mapped to a decimal datatype. In PowerBuilder, the precision of a decimal is 18 digits. If a column's datatype has a higher precision, for example number(32,30), inserting a number with a precision greater than 18 digits produces an incorrect result when the number is retrieved in a DataWindow. For example, 1.8E-17 displays as 0.000000000000000018, whereas 1.5E-25 displays as 0.

You might be able to avoid this problem by using a different datatype, such as float, for high precision number columns in the Oracle DBMS. The float datatype is mapped to the number datatype within the DataWindow's source.

Basic software components for Oracle

You must install the software components in Figure 11-1 to access an Oracle database in PowerBuilder.

Figure 11-1: Components of an Oracle connection



Preparing to use the Oracle database

Before you define the database interface and connect to an Oracle database in PowerBuilder, follow these steps to prepare the database for use:

1. Install and configure the required database server, network, and client software.
2. Install the native Oracle database interface for the version of Oracle you want to access.
3. Verify that you can connect to the Oracle server and database outside PowerBuilder.
4. (ORA driver only) Determine whether you want to use connection pooling or session pooling.

Preparing an Oracle database for use with PowerBuilder involves these basic tasks.

Step 1: Install and configure the database server

You must install and configure the database server, network, and client software for Oracle.

❖ To install and configure the database server, network, and client software:

1. Make sure the Oracle database software is installed on your computer or on the server specified in your database profile.

   For example, with the Oracle O90 interface you can access an Oracle9*i* or Oracle 10*g* database server.

   You must obtain the database server software from Oracle Corporation.

   For installation instructions, see your Oracle documentation.

2. Make sure the supported network software (such as TCP/IP) is installed and running on your computer and is properly configured so that you can connect to the Oracle database server at your site.

   The Hosts and Services files must be present on your computer and properly configured for your environment.

   You must obtain the network software from your network vendor or database vendor.

   For installation and configuration instructions, see your network or database administrator.

3. Install the required Oracle client software on each client computer on which PowerBuilder is installed.

   You must obtain the client software from Oracle Corporation. Make sure the client software version you install supports *all* of the following:

   - The operating system running on the client computer
   - The version of the database that you want to access
   - The version of PowerBuilder that you are running

   Oracle 10*g* Instant Client is free client software that lets you run applications without installing the standard Oracle client software. It has a small footprint and can be freely redistributed.

4. Make sure the Oracle client software is properly configured so that you can connect to the Oracle database server at your site.

   For information about setting up Oracle configuration files, see your Oracle Net documentation.

5. If required by your operating system, make sure the directory containing the Oracle client software is in your system path.

Step 2: Install the database interface

In the PowerBuilder Setup program, select the Typical install or select the Custom install and select the Oracle database interfaces you require.

For a list of the Oracle database interfaces available, see Supported versions for Oracle on page 133.

Step 3: Verify the connection

Make sure you can connect to the Oracle database server and log in to the database you want to access from outside PowerBuilder.

Some possible ways to verify the connection are by running the following Oracle tools:

- **Accessing the database server** Tools such as Oracle TNSPING (or any other ping utility) check whether you can reach the database server from your computer.
- **Accessing the database** Tools such as Oracle SQL*Plus check whether you can log in to the Oracle database you want to access and perform database operations. It is a good idea to specify the same connection parameters you plan to use in your PowerBuilder database profile to access the database.

Step 4: Determine whether to use connection or session pooling

Oracle client interface (OCI) pooling for PowerBuilder applications is created when you connect to an Oracle server for the first time. The pooling is identified by the server name and character set which are passed in the DBPARM parameters SQLCA.ServerName and NLS_Charset, respectively. If two Oracle connections are connected to the same Oracle server but use different character sets, the connections must reside in different connection or session pools. All pooling-related DBPARM parameters must be set before the initial database connection from PowerBuilder.

Session pooling means that the application creates and maintains a group of stateless sessions to the database. These sessions are passed to thin clients as requested. If no session is available, a new one is created. When the client is done with the session, the client releases it to the pool. With session pooling, the number of sessions in the pool can increase dynamically.

Session pooling does not support external authentication using an OS account. If a Login ID is not specified in a database connection using an existing session pool, the Login ID of the session pooling creator is used for the connection.

CNNPool parameter maintained for backward compatibiity
The O90 and O10 database drivers that you can use in PowerBuilder to connect to the 9.x and 10.x versions of the Oracle DBMS support connection pooling with the DBPARM parameter CNNPool. For backward compatibility purposes, this parameter is also supported by the ORA driver that you use with Oracle 11*g*. However, if the Pooling parameter is used with this driver, the CNNPool parameter is ignored.

Deciding on pooling type Table 11-3 describes the circumstances under which you should make your pooling selection.

Table 11-3: Pooling types and when or when not to use them

Choose	When database sessions are
Session pooling	Stateless (reusable by middle tier threads) and the number of back-end server processes can cause database scaling problems.
Connection pooling	Stateful (not reusable by middle tier threads) and the number of back-end server processes can cause database scaling problems. The number of physical connections and back-end server processes is reduced by using connection pooling. Therefore many more database sessions can be utilized for the same back-end server configuration.

Choose	When database sessions are
No pooling	Stateful (not reusable by middle tier threads) and the number of back-end server processes will never be large enough to cause scaling issues for the database. MTS components do not support either type of pooling for Oracle databases.

Setting pooling parameters The database profile dialog box for an Oracle 11*g* connection includes a Pooling tab that lets you select the pooling parameters listed in Table 11-4.

Table 11-4: Pooling parameters for the ORA driver

Pooling parameter	Description
Pooling Type	You can select Session Pooling, Connection Pooling, or None (default). Sets the Pooling DBPARM.
Runtime Connection Load Balancing	This check box selected by default. It is ignored when you select Connection Pooling or None for the Pooling Type. Sets the RTConnBalancing DBPARM.
Homogeneous Session	This check box is not selected by default and is valid for session pooling only. When selected, all sessions in the pool are authenticated with the user name and password in effect when the session pool was created. The user name and password in later connection requests are ignored. Proxy sessions cannot be created in homogeneous sessioon mode. Sets the SessionHomogeneous DBPARM.
Minimum Number of Sessions	Integer for the minimum number of database connection sessions; value is 1 by default. Sets the CSMin DBPARM. This value is ignored when the SessionHomogeneous DBPARM is set to false.
Maximum Number of Sessions	Integer for the maximum number of database connection sessions; value is 100 by default. Sets the CSMax DBPARM.
Increment	Integer for database connection increments per session; value is 1 by default. Sets the CSIncr DBPARM. This value is ignored when the SessionHomogeneous DBPARM is set to false.
Pool Creator	User name used to create the connection or session pool when the pool is not already created. Sets the PoolCreator DBParm to a string for the user name prior to the database connection. If you do not provide a value for the PoolCreator DBParm, the Transaction object's LogID and LogPass properties are used to create the pooling.

Pooling parameter	Description
Password	Password used to create the connection or session pool when the pool is not already created. Sets the PoolPwd DBParm to a string for the password for the pool creator.

Defining the Oracle database interface

To define a connection through an Oracle database interface, you must create a database profile by supplying values for at least the basic connection parameters in the Database Profile Setup dialog box for your Oracle interface. You can then select this profile at any time to connect to your database in the development environment.

For information on how to define a database profile, see Using database profiles on page 6.

Specifying the Oracle server connect descriptor

To connect to an Oracle database server that resides on a network, you must specify the proper connect descriptor in the Server box on the Connection tab of the Database Profile Setup dialog box for your Oracle interface. The connect descriptor specifies the connection parameters that Oracle uses to access the database.

For help determining the proper connect descriptor for your environment, see your Oracle documentation or system administrator.

Specifying a connect descriptor

The syntax of the connect descriptor depends on the Oracle client software you are using.

If you are using Net9 or later, the syntax is:

OracleServiceName

If you are using SQL*Net version 2.x, the syntax is:

@ TNS: *OracleServiceName*

Parameter	Description
@	The at (@) sign is required
TNS	The identifier for the Oracle Transparent Network Substrate (TNS) technology

Parameter	Description
:	The colon (:) is required
OracleServiceName	The service name assigned to your server in the Oracle configuration file for your platform

Net9 example To use Net9 client software to connect to the service named ORA9, type the following connect descriptor in the Server box on the Connection tab of the Database Profile Setup dialog box for Oracle9*i* and later: `ORA9`.

Using Oracle stored procedures as a data source

This section describes how you can use Oracle stored procedures.

What is an Oracle stored procedure?

Oracle defines a **stored procedure** (or function) as a named PL/SQL program unit that logically groups a set of SQL and other PL/SQL programming language statements together to perform a specific task.

Stored procedures can take parameters and return one or more result sets (also called cursor variables). You create stored procedures in your schema and store them in the data dictionary for use by multiple users.

What you can do with Oracle stored procedures

Ways to use Oracle stored procedures

You can use an Oracle stored procedure in the following ways in your PowerBuilder application:

- As a data source for DataWindow objects
- Called by an embedded SQL DECLARE PROCEDURE statement in a PowerBuilder application (includes support for fetching against stored procedures with result sets)
- Called as an external function or subroutine in a PowerBuilder application by using the RPCFUNC keyword when you declare the procedure

For information about the syntax for using the DECLARE PROCEDURE statement with the RPCFUNC keyword, see the *PowerScript Reference*.

Procedures with a single result set You can use stored procedures that return a single result set in reports and embedded SQL, but *not* when using the RPCFUNC keyword to declare the stored procedure as an external function or subroutine.

Procedures with multiple result sets You can use procedures that return multiple result sets *only* in embedded SQL. Multiple result sets are *not supported* in DataWindows, reports, or with the RPCFUNC keyword.

Using Oracle stored procedures with result sets

Overview of basic steps

The following procedure assumes you are creating the stored procedure in the ISQL view of the Database painter in PowerBuilder.

❖ **To use an Oracle stored procedure with a result set:**

1. Set up the ISQL view of the Database painter to create the stored procedure.
2. Create the stored procedure with a result set as an IN OUT (reference) parameter.
3. Create reports that use the stored procedure as a data source.

Setting up the Database painter

When you create a stored procedure in the ISQL view of the Database painter, you must change the default SQL statement terminator character to one that you do not plan to use in your stored procedure syntax.

The default SQL terminator character for the Database painter is a semicolon (;). If you plan to use a semicolon in your Oracle stored procedure syntax, you must change the painter’s terminator character to something other than a semicolon to avoid conflicts. A good choice is the backquote (`) character.

❖ **To change the default SQL terminator character in the Database painter:**

1. Connect to your Oracle database in PowerBuilder as the System user.

   For instructions, see Defining the Oracle database interface on page 142.
2. Open the Database painter.
3. Select Design>Options from the menu bar.

   The Database Preferences dialog box displays. If necessary, click the General tab to display the General property page.

4 Type the character you want (for example, a backquote) in the SQL Terminator Character box.

5 Click Apply or OK.

The SQL Terminator Character setting is applied to the current connection and all future connections (until you change it).

Creating the stored procedure

After setting up the Database painter, you can create an Oracle stored procedure that has a result set as an IN OUT (reference) parameter. PowerBuilder retrieves the result set to populate a report.

There are many ways to create stored procedures with result sets. The following procedure describes one possible method that you can use.

For information about when you can use stored procedures with single and multiple result sets, see What you can do with Oracle stored procedures on page 143.

❖ To create Oracle stored procedures with result sets:

1 Make sure your Oracle user account has the necessary database access and privileges to access Oracle objects (such as tables and procedures).

Without the appropriate access and privileges, you will be unable to create Oracle stored procedures.

2 Assume the following table named tt exists in your Oracle database:

a	b	c
1	Newman	sysdate
2	Everett	sysdate

3 Create an Oracle package that holds the result set type and stored procedure. The result type must match your table definition.

For example, the following statement creates an Oracle package named spm that holds a result set type named rctl and a stored procedure named proc1. The tt%ROWTYPE attribute defines rctl to contain all of the columns in table tt. The procedure proc1 takes one parameter, a cursor variable named rc1 that is an IN OUT parameter of type rctl.

```
CREATE OR REPLACE PACKAGE spm
   IS TYPE rctl IS REF CURSOR
   RETURN tt%ROWTYPE;
   PROCEDURE proc1(rc1 IN OUT rctl);END;`
```

4 Create the Oracle stored procedure separately from the package you defined.

The following examples show how to create two stored procedures: *spm_proc 1* (returns a single result set) and spm_proc2 (returns multiple result sets).

The IN OUT specification means that PowerBuilder passes the cursor variable (rc1 or rc2) by reference to the Oracle procedure and expects the procedure to open the cursor. After the procedure call, PowerBuilder fetches the result set from the cursor and then closes the cursor.

spm_proc1 example for reports The following statements create spm_proc1 which returns one result set. You can use this procedure as the data source for a report in PowerBuilder.

```
CREATE OR REPLACE PROCEDURE spm_proc1(rc1 IN OUT
spm.rctl)
AS
BEGIN
   OPEN rc1 FOR SELECT * FROM tt;
END;`
```

spm_proc2 example for embedded SQL The following statements create spm_proc2 which returns two result sets. You can use this procedure only in embedded SQL.

```
CREATE OR REPLACE PROCEDURE spm_proc2 (rc1 IN OUT
spm.rctl, rc2 IN OUT spm.rctl)
AS
BEGIN
   OPEN rc1 FOR SELECT * FROM tt ORDER BY 1;
   OPEN rc2 FOR SELECT * FROM tt ORDER BY 2;END;`
```

Error checking
If necessary, check the Oracle system table public.user_errors for a list of errors.

Creating the report

After you create the stored procedure, you can define the report that uses the stored procedure as a data source.

You can use Oracle stored procedures that return a single result set in a report. If your stored procedure returns multiple result sets, you must use embedded SQL commands to access it.

The following procedure assumes that your Oracle stored procedure returns only a single result set.

❖ **To create a report using an Oracle stored procedure with a result set:**

1 Select a presentation style on the DataWindow page of the New dialog box and click OK.

2 Select the Stored Procedure icon and click OK.

The Select Stored Procedure wizard page displays, listing the stored procedures available in your database.

3 Select the stored procedure you want to use as a data source, and click Next.

4 Complete the wizard to define the report.

When you preview the report or call Retrieve, PowerBuilder fetches the result set from the cursor in order to populate the report. If you selected Retrieve on Preview on the Choose Data Source page in the wizard, the result set displays in the Preview view when the DataWindow opens.

Using a large-object output parameter

You can define a large object (LOB) as an output parameter for an Oracle stored procedure or function to retrieve large-object data. There is no limit on the number of LOB output arguments that can be defined for each stored procedure or function.

In Oracle 10*g*, the maximum size of LOB datatypes has been increased from 4 gigabytes minus 1 to 4 gigabytes minus 1 multiplied by the block size of the database. For a database with a block size of 32K, the maximum size is 128 terabytes.

RPC calls to stored procedures with array parameters

If your application performs a remote procedure call (RPC) that passes an array parameter to an Oracle stored procedure, the array size in the stored procedure must not be zero. If the array size is uninitialized (has no size), the PBVM returns an error.

Using Oracle user-defined types

PowerBuilder supports SQL CREATE TYPE and CREATE TABLE statements for Oracle user-defined types (objects) in the ISQL view of the Database painter. It correctly handles SQL SELECT, INSERT, UPDATE, and DELETE statements for user-defined types in the Database and Report painters.

This means that using the Oracle native database interfaces in PowerBuilder, you can:

Do this	In
Use Oracle syntax to create user-defined types	Database painter
Use Oracle syntax to create tables with columns that reference user-defined types	Database painter
View columns in Oracle tables that reference user-defined types	Database painter
Manipulate data in Oracle tables that have user-defined types	Database painter Report painter DataWindow objects
Export Oracle table syntax containing user-defined types to a log file	Database painter
Invoke methods	Report painter (Compute tab in SQL Toolbox)

Example

Here is a simple example that shows how you might create and use Oracle user-defined types in PowerBuilder.

For more information about Oracle user-defined types, see your Oracle documentation.

❖ **To create and use Oracle user-defined types:**

1 In the ISQL view of the Database painter, create two Oracle user-defined types: ball_stats_type and player_type.

Here is the Oracle syntax to create ball_stats_type. Notice that the ball_stats object of type ball_stats_type has a method associated with it called get_avg.

```
CREATE OR REPLACE TYPE ball_stats_type AS OBJECT
(bat_avg NUMBER(4,3),rbi NUMBER(3),MEMBER FUNCTION
get_avg RETURN NUMBER,PRAGMA RESTRICT_REFERENCES
(get_avg,WNDS,RNPS,WNPS));
CREATE OR REPLACE TYPE BODY ball_stats_type ASMEMBER
FUNCTION get_avg RETURN NUMBER ISBEGINRETURN
SELF.bat_avg;
END;
END;
```

Here is the Oracle SQL syntax to create player_type. Player_type references the user-defined type ball_stats_type. PowerBuilder supports such nesting graphically in the Database, Report, and Table painters (see step 3).

```
CREATE TYPE player_type AS OBJECT (player_no
NUMBER(2),player_name VARCHAR2(30),ball_stats
ball_stats_type);
```

2 In the Database painter, create a table named lineup that references these user-defined types.

Here is the Oracle SQL syntax to create the lineup table and insert a row. Lineup references the player_type user-defined type.

```
CREATE TABLE lineup (position NUMBER(2) NOT NULL,
player player_type);
INSERT INTO lineup VALUES (1,player_type (15,
'Dustin Pedroia', ball_stats_type (0.317, 50)));
```

3 Display the lineup table in the Database or Report painter.

PowerBuilder uses the following structure->member notation to display the table:

```
lineup
======
position
player->player_no
player->player_name
player->ball_stats->bat_avg
player->ball_stats->rbi
```

4. To access the get_avg method of the object ball_stats contained in the object column player, use the following structure->member notation when defining a computed column for the report. For example, when working in the Report painter, you could use this notation on the Compute tab in the SQL Toolbox:

```
player->ball_stats->get_avg()
```

Support for HA event notification

Oracle Real Application Clusters (RAC) is a cluster database that uses a shared cache architecture. In Oracle 10*g* Release 2, a High Availability (HA) client connected to an RAC database can register a callback to indicate that it wants the server to notify it in case of a database failure event that affects a connection made by the client.

To take advantage of this feature, PowerBuilder users can script the DBNotification event of the Transaction object. For more information, see the description of the DBNotification event and the HANotification database parameter in the online Help.

ORA driver support for Oracle 11g features

In addition to support for Oracle 11*g* session pooling and connection pooling, the ORA driver adds support for other 11*g* features.

Client result cache

The PowerBuilder ORA driver supports Oracle Client Cache, however this feature depends on your Oracle Server and Client configuration. You can configure the Oracle Client Cache with an *init.ora* or *sqlnet.ora* file. Cached queries are annotated with "`/*+ result_cache */`" hints to indicate that results are stored in the query result cache. You enable OCI statement caching from PowerBuilder applications with the StatementCache DBPARM parameter.

Application driver name

An OCI application can choose its own name and set it as a diagnostic aid. The AppDriverName DPBARM parameter allows you to set your own client driver name for the PowerBuilder ORA interface. The maximum length of the name is 8 characters. You can display the client driver name with the V$SESSION_CONNECT_INFO or GV$SESSION_CONNECT_INFO dynamic performance view queries.

Client access through a proxy (Oracle 10.2 feature)

The PowerBuilder ORA driver supports the proxy authentication feature that was introduced in Oracle 10.2. With proxy authentication, the end user typically authenticates to a middle tier (such as a firewall), that in turn logs into the database on the user's behalf—as a proxy user. After logging into the database, the proxy user can switch to the end user's identity and perform operations using the authorization accorded to that user.

The ConnectAs DBParm parameter allows you to take advantage of this proxy connection feature. For example, if the user’s Transaction object LogID is “Scott” and you set the ConnectAs DBParm parameter to “John”, the OCI client logs in to database as the proxy user (“Scott”), then switches to the end user identity (“John”).

If you are using connection or session pooling, the proxy user name is the connection or session pooling creator (which you can provide in the PoolCreator and PoolPwd DBParm parameters), and the Transaction object’s LogID is ignored. No proxy session can be created if pooling is set to HomogeneousSession mode.

Limitation on proxy connection without pooling
When using a proxy connection without pooling, you must set the NLS_Charset DBPARM to “Local” or to another non-Unicode character set. If you do not change the “Unicode” default value for this DBPARM, the connection fails because the Oracle Client Interface does not accept a Unicode name string for its proxy client attribute.

Load balancing

The Oracle Real Application Clusters (RAC) database option allows a single database to be hosted in multiple instances on multiple nodes of the database server. This adds high availability and failover capacity to the database. Availability is improved since, if one node fails, another node can assume its workload. All instances have access to the whole database. The shared disk method of clustering databases used by the RAC option increases scalability because nodes can be added or freed as required.

In RAC environments, session pools can use service metrics received from the RAC load balancing advisory to balance application session requests. The work requests coming into the session pool can then be distributed across the instances of RAC based on current service performance.

Connect time load balancing Balancing of work requests occur at two different times: connect time and runtime. Connect time load balancing occurs when a session is first created by the application. This ensures that sessions that are part of the pool are well distributed across RAC instances, and that sessions on each of the instances get a chance to execute work.

For session pools that support services at one instance only, the first available session in the pool is adequate. When the pool supports services that span multiple instances, there is a need to distribute the work requests across instances so that the instances that are providing better service or have greater capacity get more requests.

Runtime connection load balancing Runtime connection load balancing basically routs work requests to the sessions in a session pool that best serve the work. Runtime connection load balancing is enabled by default when an Oracle 11.1 or higher client is connected to a 10.2 or higher Oracle server using OCI session pooling.

The DBPARM parameter, RTConnBalancing, supports the runtime connection load balancing feature. It is available only when the Pooling parameter is set to Session Pooling, and it can be set before connection only. By default, when you select Session Pooling for the pooling type, the RTConnBalancing value is true.

CHAPTER 12 Using DirectConnect

About this chapter

This chapter describes how to use the DirectConnect™ interface in PowerBuilder.

Contents

Using the DirectConnect interface

The DirectConnect interface uses SAP's Open Client CT-Library (CT-Lib) API to access a database through SAP middleware data access products such as the DirectConnect for OS/390 component of Mainframe Connect™ and Open ServerConnect™.

Accessing Unicode data

PowerBuilder can connect, save, and retrieve data in both ANSI/DBCS and Unicode databases. When character data or command text is sent to the database, PowerBuilder sends a DBCS string if the UTF8 database parameter is set to 0 (the default). If UTF8 is set to 1, PowerBuilder sends a UTF-8 string.

The database server must have the UTF-8 character set installed. See the description of the UTF-8 database parameter in the online Help for more information.

A Unicode database is a database whose character set is set to a Unicode format, such as UTF-8, UTF-16, UCS-2, or UCS-4. All data must be in Unicode format, and any data saved to the database must be converted to Unicode data implicitly or explicitly.

A database that uses ANSI (or DBCS) as its character set might use special datatypes to store Unicode data. Columns with these datatypes can store *only* Unicode data. Any data saved into such a column must be converted to Unicode explicitly. This conversion must be handled by the database server or client.

Connecting through the DirectConnect middleware product

SAP DirectConnect is a data access server that provides a standardized middleware interface between your applications and your enterprise data sources. Data access services to a particular database are defined in a DirectConnect server. Since a DirectConnect server can support multiple access services, you can access multiple databases through a single server.

When you use the DirectConnect interface to connect to a particular database, your connection is routed through the access service for that database. An access service consists of a named set of configuration properties and a specific access service library.

To access DB2 data on an IBM mainframe through a DirectConnect server, you can use the DirectConnect interface to connect through either a DirectConnect for MVS access service or a DirectConnect Transaction Router Service (TRS).

TRS provides fast access to a DB2/MVS database by using remote stored procedures. The DirectConnect interface supports both versions of the TRS library: TRSLU62 and TRSTCP.

The DirectConnect server operates in two modes: SQL transformation and passthrough. The DirectConnect interface for DB2/MVS uses passthrough mode, which allows your PowerBuilder application to have direct access to the capabilities of the DB2/MVS data source.

Connecting through the Open ServerConnect middleware product

SAP's Open ServerConnect supports mainframe applications that retrieve and update data stored on the mainframe that SAP client applications can execute. Client applications can connect directly to a DB2/MVS database through an Open ServerConnect application residing on the mainframe, eliminating the need for an intermediate gateway like DirectConnect. (This type of connection is also known as a *gateway-less* connection.) In addition, an Open ServerConnect application presents mainframe Remote Procedure Calls (RPCs) as database stored procedures to the client application.

To access DB2 data on an IBM mainframe through Open ServerConnect, you can use the DirectConnect interface to connect through Open ServerConnect for IMS and MVS.

Selecting the type of connection

To select how PowerBuilder accesses the database, use the Choose Gateway drop-down list on the Connection tab of the DirectConnect Database Profile Setup dialog box and select one of the following:

- Access Service
- Gatewayless
- TRS

All the DBParm parameters defined for the DirectConnect interface are applicable to all three connections except the following:

- HostReqOwner applies to Access Service and Gatewayless only
- Request, ShowWarnings, and SystemOwner apply to Access Service only
- UseProcSyntax applies to Gatewayless only

See the online help for the complete list of DBParm parameters applicable to the DirectConnect interface.

Supported versions for the DirectConnect interface

The DirectConnect interface uses a DLL named *PBDIR170.DLL* to access a database through either DirectConnect or Open ServerConnect.

Required DirectConnect versions

To access a DB2/MVS database through the access service, it is strongly recommended that you use DirectConnect for MVS access service version 11.1.1p4 or later.

To access a DB2/MVS database through TRS, it is strongly recommended that you use DirectConnect TRS version 11.1.1p4 or later.

For information on DirectConnect for MVS and TRS, see your DirectConnect documentation.

Required Open ServerConnect versions

To access a DB2/MVS database through Open ServerConnect, it is strongly recommended that you use Open ServerConnect IMS and MVS version 4.0 or later.

For information on Open ServerConnect for MVS, see your Open ServerConnect documentation.

Supported DirectConnect interface datatypes

The DirectConnect interface supports the PowerBuilder datatypes listed in Table 12-1 in reports. and embedded SQL.

Table 12-1: Supported datatypes for DirectConnect

Char (fewer than 255 characters)	Long VarChar
Char for Bit Data	Real
Date	SmallInt
Decimal	Time
Double Precision	Timestamp (DateTime)
Float	VarChar
Integer	VarChar for Bit Data

Basic software components for the DirectConnect interface

Figure 12-1 shows the basic software components required to access a database using the DirectConnect interface and the DirectConnect middleware data access product.

Figure 12-1: Components of a DirectConnect connection using DirectConnect middleware

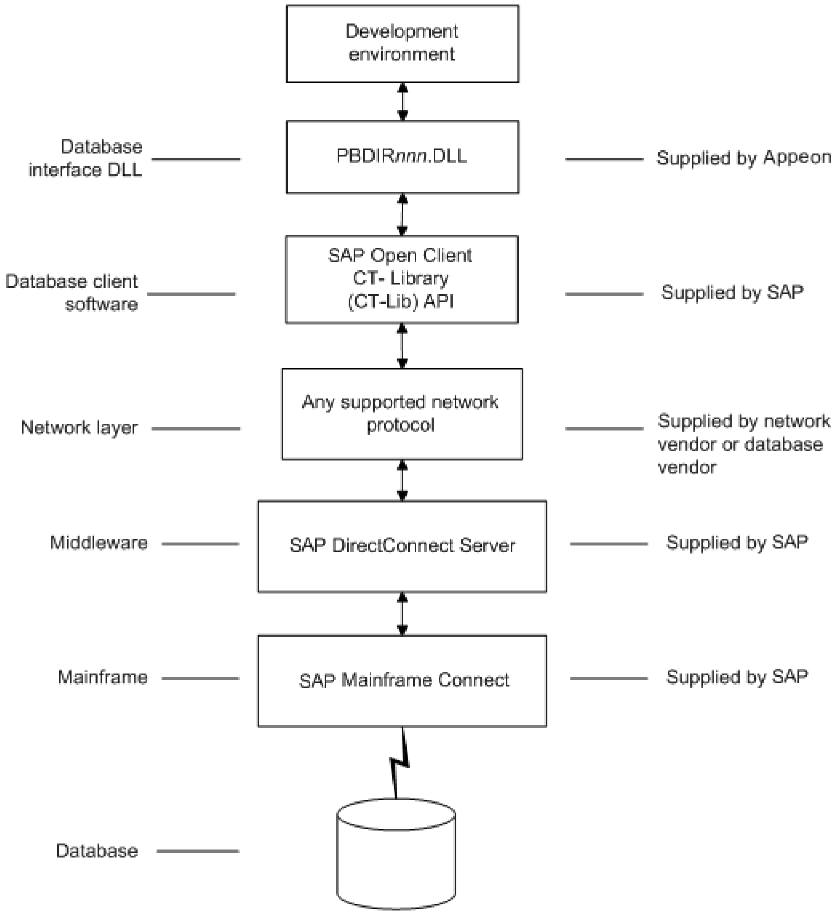
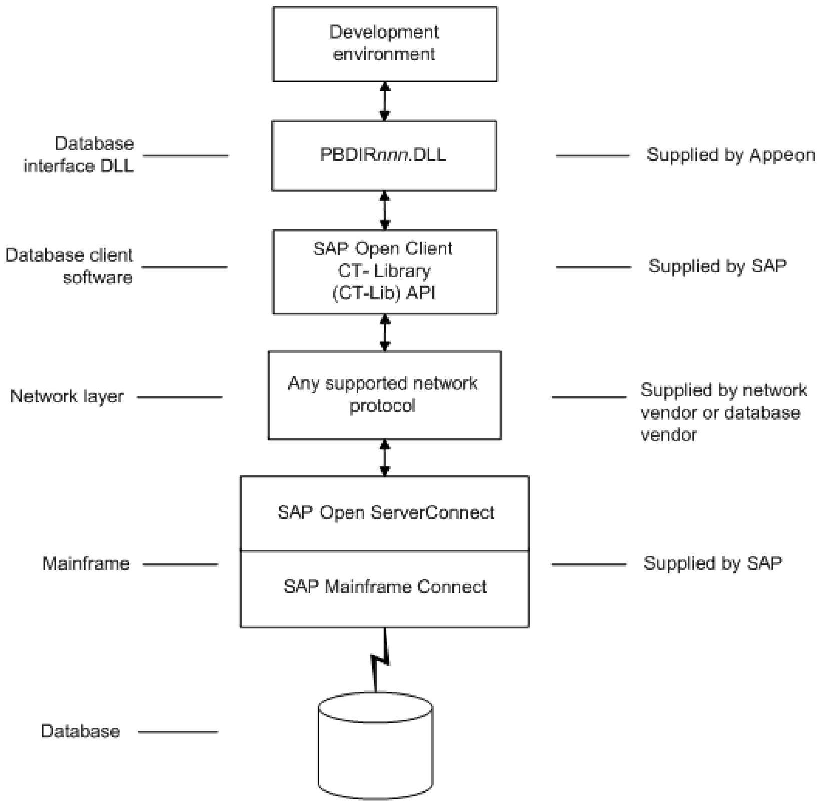


Figure 12-2 shows the basic software components required to access a database using the DirectConnect interface and the Open ServerConnect middleware data access product.

Figure 12-2: Components of a DirectConnect connection using Open ServerConnect middleware



Preparing to use the database with DirectConnect

Before you define the interface and connect to a database through the DirectConnect interface, follow these steps to prepare the database for use:

1 Install and configure the SAP middleware data access products, network, and client software.

2 Install the DirectConnect interface.

3 Verify that you can connect to your middleware product and your database outside PowerBuilder.

4 Create the extended attribute system tables outside PowerBuilder.

Step 1: Install and configure the SAP middleware product

You must install and configure the SAP middleware data access product, network, and client software.

❖ To install and configure the SAP middleware data access product, network, and client software:

1 Make sure the appropriate database software is installed and running on its server.

   You must obtain the database server software from your database vendor.

   For installation instructions, see your database vendor's documentation.

2 Make sure the appropriate DirectConnect access service software is installed and running on the DirectConnect server specified in your database profile
   or
   Make sure the appropriate Open ServerConnect software is installed and running on the mainframe specified in your database profile.

3 Make sure the required network software (such as TCP/IP) is installed and running on your computer and is properly configured so you that can connect to the DirectConnect server or mainframe at your site.

   You must install the network communication driver that supports the network protocol and operating system platform you are using.

   For installation and configuration instructions, see your network or database administrator.

4 Install the required Open Client CT-Library (CT-Lib) software on each client computer on which PowerBuilder is installed.

   You must obtain the Open Client software from SAP. Make sure the version of Open Client you install supports *both* of the following:

   The operating system running on the client computer
   The version of PowerBuilder that you are running

Open Client required
To use the DirectConnect interface, you must install Open Client.

For information about Open Client, see your Open Client documentation.

5. Make sure the Open Client software is properly configured so you can connect to the middleware data access product at your site.

   Installing the Open Client software places the *SQL.INI* configuration file in the SQL Server directory on your computer. *SQL.INI* provides information that SQL Server uses to find and connect to the middleware product at your site. You can enter and modify information in *SQL.INI* with the configuration utility or editor that comes with the Open Client software.

   For information about editing the *SQL.INI* file, see Editing the SQL.INI file on page 161. For more information about setting up *SQL.INI* or any other required configuration file, see your SQL Server documentation.

6. If required by your operating system, make sure the directory containing the Open Client software is in your system path.

7. Make sure only one copy of each of the following files is installed on your client computer:

   - DirectConnect interface DLL
   - Network communication DLL (such as *NLWNSCK.DLL* for Windows Sockets-compliant TCP/IP)
   - Open Client DLLs (such as *LIBCT.DLL* and *LIBCS.DLL*)

Step 2: Install the interface

In the PowerBuilder Setup program, select the Typical install, or select the Custom install and select the Direct Connect Interface (DIR).

Step 3: Verify the connection

Make sure you can connect to your middleware product and your database and log in to the database you want to access from outside PowerBuilder.

Some possible ways to verify the connection are by running the following tools:

- **Accessing the database server** Tools such as the Open Client/Open Server Configuration utility (or any Ping utility) check whether you can reach the database server from your computer.
- **Accessing the database** Tools such as ISQL or SQL Advantage (interactive SQL utilities) check whether you can log in to the database and perform database operations. It is a good idea to specify the same connection parameters you plan to use in your PowerBuilder database profile to access the database.

Step 4: Create the extended attribute system tables

PowerBuilder uses a collection of five system tables to store extended attribute information. When using the DirectConnect interface, you *must* create the extended attribute system tables outside PowerBuilder to control the access rights and location of these tables.

Run the *DB2SYSPB.SQL* script outside PowerBuilder using the SQL tool of your choice.

For instructions, see Creating the extended attribute system tables in DB2 databases on page 162.

Editing the SQL.INI file

Make sure the *SQL.INI* file provides an entry about either the access service being used and the DirectConnect server on which it resides or the Open ServerConnect program being used and the mainframe on which it resides.

For the server object name, you need to provide the exact access service name as it is defined in the access service library configuration file on the DirectConnect server. You must also specify the network communication DLL being used, the TCP/IP address or alias used for the DirectConnect server on which the access service resides, and the port on which the DirectConnect server listens for requests:

```
[access_service_name]
query=network_dll,server_alias,server_port_no
```

PowerBuilder users must also specify the access service name in the SQLCA.ServerName property of the Transaction object.

Defining the DirectConnect interface

To define a connection through the DirectConnect interface, you must create a database profile by supplying values for at least the basic connection parameters in the Database Profile Setup - DirectConnect dialog box. You can then select this profile anytime to connect to your database in the development environment.

For information on how to define a database profile, see Using database profiles on page 6.

Creating the extended attribute system tables in DB2 databases

This section describes how PowerBuilder creates the extended attribute system tables in your DB2 database to store extended attribute information. It then explains how to use the *DB2SYSPB.SQL* script to create the extended attribute system tables outside PowerBuilder.

You can use the *DB2SYSPB.SQL* script if you are connecting to the IBM DB2 family of databases through any of the following database interfaces:

- ODBC interface
- SAP DirectConnect interface

Creating the extended attribute system tables

When you create or modify a table in PowerBuilder, the information you provide is stored in five system tables in your database. These system tables contain extended attribute information such as the text to use for labels and column headings, validation rules, display formats, and edit styles. (These system tables are different from the system tables provided by your DB2 database.)

By default, the extended attribute system tables are created automatically the first time a user connects to the database using PowerBuilder.

When you use the DirectConnect interface
When you use the DirectConnect interface, the extended attribute system tables are *not* created automatically. You must run the *DB2SYSPB.SQL* script to create the system tables as described in Using the DB2SYSPB.SQL script on page 163.

❖ **To ensure that the extended attribute system tables are created with the proper access rights:**

- Make sure the first person to connect to the database with PowerBuilder has sufficient authority to create tables and grant permissions to PUBLIC.

  This means that the first person to connect to the database should log in as the database owner, database administrator, system user, system administrator, or system owner, as specified by your DBMS.

Using the DB2SYSPB.SQL script

Why do this

If you are a system administrator at a DB2 site, you might prefer to create the extended attribute system tables outside PowerBuilder for two reasons:

- The first user to connect to the DB2 database using PowerBuilder might not have the proper authority to create tables.
- When PowerBuilder creates the extended attribute system tables, it places them in the default tablespace. This might not be appropriate for your needs.

When using the DirectConnect interface
You *must* create the extended attribute system tables outside PowerBuilder if you are using the DirectConnect interface. You need to decide which database and tablespace should store the system tables. You might also want to grant update privileges only to specific developers or groups.

What you do

To create the extended attribute system tables, you run the *DB2SYSPB.SQL* script outside PowerBuilder. This script contains SQL commands that create and initialize the system tables with the table owner and tablespace you specify.

Where to find DB2SYSPB.SQL

The *DB2SYSPB.SQL* script is in the *Server* directory in the PowerBuilder setup program. This directory contains server-side installation components and is *not installed* with PowerBuilder on your computer.

You can access the *DB2SYSPB.SQL* script by copying it to your computer.

Use the following procedure *from the database server* to create the extended attribute system tables in a DB2 database outside PowerBuilder. This procedure assumes you are accessing the *DB2SYSPB.SQL* script from *d:\server*.

❖ **To create the extended attribute system tables in a DB2 database outside PowerBuilder:**

1. Log in to the database server or gateway as the system administrator.
2. Use any text editor to modify *d:\server\DB2SYSPB.SQL* for your environment. You can do any of the following:
   - Change all instances of PBOwner to another name.

     Specifying SYSIBM is prohibited
     You cannot specify SYSIBM as the table owner. This is prohibited by DB2.

- Change all instances of database.tablespace to the appropriate value.
- Add appropriate SQL statement delimiters for the tool you are using to run the script.
- Remove comments and blank lines if necessary.

PBCatalogOwner
If you changed PBOwner to another name in the *DB2SYSPB.SQL* script, you must specify the new owner name as the value for the PBCatalogOwner DBParm parameter in your database profile. For instructions, see PBCatalogOwner in the online Help.

3. Save any changes you made to the *DB2SYSPB.SQL* script.
4. Execute the *DB2SYSPB.SQL* script from the database server or gateway using the SQL tool of your choice.

PART 4

Working with Database Connections

This part describes how to establish, manage, and troubleshoot database connections.

CHAPTER 13 Managing Database Connections

About this chapter

After you install the necessary database software and define the database interface, you can connect to the database from PowerBuilder. Once you connect to the database, you can work with the tables and views stored in that database.

This chapter describes how to connect to a database in PowerBuilder, maintain database profiles, and share database profiles.

Contents

Terminology

In this chapter, the term **database** refers to *both* of the following unless otherwise specified:

- A database or DBMS that you access with a standard database interface and appropriate driver
- A database or DBMS that you access with the appropriate native database interface

About database connections

This section gives an overview of when database connections occur in PowerBuilder. It also explains why you should use database profiles to manage your database connections.

When database connections occur

Connections in PowerBuilder

PowerBuilder connects to your database when you:

- Open a painter that accesses the database
- Compile or save a PowerBuilder script containing embedded SQL statements (such as a CONNECT statement)
- Execute an application that accesses the database
- Invoke a DataWindow control function that accesses the database while executing an application

How PowerBuilder determines which database to access

PowerBuilder *connects to the database you used last* when you open a painter that accesses the database. PowerBuilder determines which database you used last by reading a setting in the registry.

What's in this book

This book describes how to connect to your database when you are working in the PowerBuilder development environment.

For instructions on connecting to a database in a PowerBuilder application, see *Application Techniques*.

Using database profiles

What is a database profile?

A **database profile** is a named set of parameters stored in the registry that defines a connection to a particular database in the PowerBuilder development environment.

Why use database profiles?

Creating and using database profiles is the easiest way to manage your database connections in PowerBuilder because you can:

- Select a database profile to establish or change database connections. You can easily connect to another database anytime during a PowerBuilder session. This is particularly useful if you often switch between different database connections.
- Edit a database profile to modify or supply additional connection parameters.
- Use the Preview tab page to test a connection and copy the connection syntax to your application code.
- Delete a database profile if you no longer need to access that data.
- Import and export profiles.

Because database profiles are created when you define your data and are stored in the registry, they have the following benefits:

- They are always available to you.
- Connection parameters supplied in a database profile are saved until you edit or delete the database profile.

Connecting to a database

To establish or change a database connection in PowerBuilder, use a database profile. You can select the database profile for the database you want to access in the Database Profiles dialog box. For how to create a database profile, see Creating a database profile on page 9.

Using the Database painter to select a database profile
You can also select the database profile for the database you want to access from the Database painter's Objects view. However, this method requires more system resources than using the Database Profiles dialog box.

Selecting a database profile

You can select a database profile from the Database Profiles dialog box.

❖ **To connect to a database using the Database Profiles dialog box:**

1 Click the Database Profile button in the PowerBar or select Tools>Database Profile from the menu bar.

Database Profile button
If your PowerBar does not include the Database Profile button, use the customize feature to add the button to the PowerBar. Having the Database Profile button on your PowerBar is useful if you frequently switch connections between different databases. For instructions on customizing toolbars, see the *Users Guide*.

The Database Profiles dialog box displays, listing your installed database interfaces.

Where the interface list comes from
When you run the Setup program, it updates the Vendors list in the registry with the interfaces you install. The Database Profiles dialog box displays the same interfaces that appear in the Vendors list.

2. Click the plus sign (+) to the left of the interface you are using or double-click the name.

   The list expands to display the database profiles defined for your interface.

3. Select the name of the database profile you want to access and click Connect or display the pop-up menu for a database profile and select Connect.

   PowerBuilder connects to the specified database and returns you to the painter workspace.

Database painter Objects view

You can select a database profile from the Database painter Objects view.

❖ To connect to a database using the Database painter:

1. Click the Database painter button in the PowerBar.

   The Database painter displays. The Objects view lists your installed database interfaces.

Where the interface list comes from
When you run the Setup program, it updates the Vendors list in the registry with the interfaces you install. The Database painter Objects view displays the same interfaces that appear in the Vendors list.

2. Click the plus sign (+) to the left of the interface you are using or double-click the name.

   The list expands to display the database profiles defined for your interface.

3. Select the name of the database profile you want to access and click the Connect button, or display the pop-up menu for a database profile and select Connect.

What happens when you connect

When you connect to a database by selecting its database profile, PowerBuilder writes the profile name and its connection parameters to the registry key *HKEY_CURRENT_USER\Software\Sybase\PowerBuilder\17.0\DatabaseProfiles\PowerBuilder*.

Each time you connect to a different database, PowerBuilder overwrites the “most-recently used” profile name in the registry with the name for the new database connection.

When you open a painter that accesses the database, you are connected to the database you used last. PowerBuilder determines which database this is by reading the registry.

The three-letter abbreviation for the database interface followed by the name of the database profile displays in PowerBuilder's main title bar. If you are working with a report, this visual cue makes it easier to check that you are using the right connection.

For example, if you open the PowerBuilder Code Examples workspace and connect to the Demo database, the title bar displays “pbexamples - ODB [PB Demo DB V2017] - PowerBuilder.”

Specifying passwords in database profiles

Your password does *not* display when you specify it in the Database Profile Setup dialog box.

However, when PowerBuilder stores the values for this profile in the registry, the actual password *does* display, in encrypted form, in the DatabasePassword or LogPassword field.

Suppressing display in the profile registry entry

To suppress password display in the profile registry entry, do the following when you create a database profile.

❖ **To suppress password display in the profile registry entry:**

1. Select the Prompt For Database Information check box on the Connection tab in the Database Profile Setup dialog box.

   This tells PowerBuilder to prompt for any missing information when you select this profile to connect to the database.

2 Leave the Password box blank. Instead, specify the password in the dialog box that displays to prompt you for additional information when you connect to the database.

What happens When you specify the password in response to a prompt instead of in the Database Profile Setup dialog box, the password does not display in the registry entry for this profile.

For example, if you do not supply a password in the Database Profile Setup - Adaptive Server Enterprise dialog box when creating a database profile, the Client Library Login dialog box displays to prompt you for the missing information.

Using the Preview tab to connect in a PowerBuilder application

To access a database in a PowerBuilder application, you must specify the required connection parameters as properties of the Transaction object (SQLCA by default) in the appropriate script. For example, you might specify the connection parameters in the script that opens the application.

In PowerBuilder, the Preview tab in the Database Profile Setup dialog box makes it easy to generate accurate PowerScript connection syntax in the development environment for use in your PowerBuilder application script.

For instructions on using the Preview tab to help you connect in a PowerBuilder application, see the section on using Transaction objects in *Application Techniques*.

Maintaining database profiles

You can easily edit or delete an existing database profile in PowerBuilder.

You can edit a database profile to change one or more of its connection parameters. You can delete a database profile when you no longer need to access its data. You can also change a profile using either the Database Profiles dialog box or the Database painter.

What happens When you edit or delete a database profile, PowerBuilder either updates the database profile entry in the registry or removes it.

Deleting a profile for an ODBC data source
If you delete a database profile that connects to an ODBC data source, PowerBuilder does *not* delete the corresponding data source definition from the ODBC initialization file. This lets you re-create the database profile later if necessary without having to redefine the data source.

Sharing database profiles

When you work in PowerBuilder, you can share database profiles among users.

Sharing database profiles between SAP tools
Since the database profiles used by PowerBuilder and InfoMaker are stored in a common registry location, database profiles you create in any of these tools are automatically available for use by the others, if the tools are running on the same computer.

This section describes what you need to know to set up, use, and maintain shared database profiles in PowerBuilder.

About shared database profiles

You can share database profiles in the PowerBuilder development environment by specifying the location of a file containing the profiles you want to share. You specify this location in the Database Preferences dialog box in the Database painter.

Where to store a shared profile file

To share database profiles among all PowerBuilder users at your site, store a profile file on a network file server accessible to all users.

When you share database profiles, PowerBuilder displays shared database profiles from the file you specify as well as those from your registry.

Shared database profiles are read-only. You can select a shared profile to connect to a database—but you *cannot* edit, save, or delete profiles that are shared. (You can, however, make changes to a shared profile and save it on your computer, as described in Making local changes to shared database profiles on page 175.)

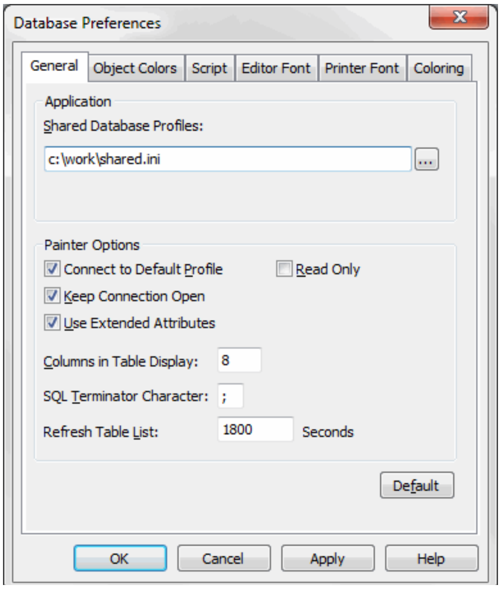
Setting up shared database profiles

You set up shared database profiles in the Database Preferences dialog box.

❖ **To set up shared database profiles:**

1. In the Database painter, select Design>Options from the menu bar to display the Database Preferences dialog box.
2. In the Shared Database Profiles box on the General tab page, specify the location of the file containing the database profiles you want to share. Do this in either of the following ways:
   - Type the location (path name) in the Shared Database Profiles box.
   - Click the Browse button to navigate to the file location and display it in the Shared Database Profiles box.

   In the following example, *c:\work\share.ini* is the location of the file containing the database profiles to be shared:



Shared Database Profiles: c:\work\shared.ini

☑ Connect to Default Profile ☐ Read Only
☑ Keep Connection Open
☑ Use Extended Attributes

Columns in Table Display: 8
SQL Terminator Character: ;
Refresh Table List: 1800 Seconds

3. Click OK.

   PowerBuilder applies the Shared Database Profiles setting to the current connection and all future connections and saves the setting in the registry.

Using shared database profiles to connect

You select a shared database profile to connect to a database the same way you select a profile stored in your registry. You can select the shared profile in the Database Profiles dialog box or from the File>Connect menu.

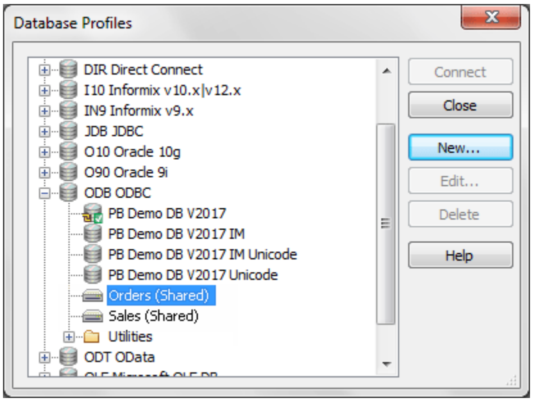
Database Profiles dialog box

You can select and connect to a shared database profile in the Database Profiles dialog box.

❖ To select a shared database profile in the Database Profiles dialog box:

1 Click the Database Profile button in the PowerBar or select Tools>Database Profile from the menu bar.

The Database Profiles dialog box displays, listing both shared and local profiles. Shared profiles are denoted by a network icon and the word *(Shared)*.



2 Select the name of the shared profile you want to access and click Connect.

PowerBuilder connects to the selected database and returns you to the painter workspace.

Making local changes to shared database profiles

Because shared database profiles can be accessed by multiple users running PowerBuilder, you should not make changes to these profiles. However, if you want to modify and save a copy of a shared database profile *for your own use*, you can edit the profile and save the modified copy in your computer's registry.

❖ **To save changes to a shared database profile in your registry:**

1. In the Database Profiles dialog box, select the shared profile you want to edit and click the Edit button.
2. In the Database Profile Setup dialog box that displays, edit the profile values as needed and click OK.

   A message box displays, asking if you want to save a copy of the modified profile to your computer.
3. Click Yes.

   PowerBuilder saves the modified profile in your computer's registry.

Maintaining shared database profiles

If you maintain the database profiles for PowerBuilder at your site, you might need to update shared database profiles from time to time and make these changes available to your users.

Because shared database profiles can be accessed by multiple users running PowerBuilder, it is *not* a good idea to make changes to the profiles over a network. Instead, you should make any changes locally and then provide the updated profiles to your users.

❖ **To maintain shared database profiles at your site:**

1. Make and save required changes to the shared profiles on your own computer. These changes are saved in your registry.

   For instructions, see Making local changes to shared database profiles on page 175.
2. Export the updated profile entries from your registry to the existing file containing shared profiles.

   For instructions, see Importing and exporting database profiles on page 177.
3. If they have not already done so, have users specify the location of the new profiles file in the Database Preferences property sheet so that they can access the updated shared profiles on their computer.

   For instructions, see Setting up shared database profiles on page 174.

Importing and exporting database profiles

Each database interface provides an Import Profile(s) and an Export Profile(s) option. You can use the Import option to import a previously defined profile for use with an installed database interface. Conversely, you can use the Export option to export a defined profile for use by another user.

The ability to import and export profiles provides a way to move profiles easily between developers. It also means you no longer have to maintain a shared file to maintain profiles. It is ideal for mobile development when you cannot rely on connecting to a network to share a file.

❖ To import a profile:

1. Highlight a database interface and select Import Profile(s) from the pop-up menu. (In the Database painter, select Import Profile(s) from the File or pop-up menu.)
2. From the Select Profile File dialog box, select the file whose profiles you want to import and click Save.
3. Select the profile(s) you want to import from the Import Profile(s) dialog box and click OK.

   The profiles are copied into your registry. If a profile with the same name already exists, you are asked if you want to overwrite it.

❖ To export a profile:

1. Highlight a database interface and select Export Profile(s) from the pop-up menu. (In the Database painter, select Export Profile(s) from the File or pop-up menu.)
2. Select the profile(s) you want to export from the Export Profile(s) dialog box and click OK.

   The Export Profile(s) dialog box lists all profiles defined in your registry regardless of the database interface for which they were defined. By default, the profiles defined for the selected database interface are marked for export.
3. From the Select Profile File dialog box, select a directory and a file in which to save the exported profile(s) and click Save.

   The exported profiles can be saved to a new or existing file. If saved to an existing file, the profile(s) are added to the existing profiles. If a profile with the same name already exists, you are asked if you want to overwrite it.

About the PowerBuilder extended attribute system tables

PowerBuilder uses a collection of five system tables to store extended attribute information (such as display formats, validation rules, and font information) about tables and columns in your database. You can also define extended attributes when you create or modify a table in PowerBuilder.

This section tells you how to:

- Make sure the PowerBuilder extended attribute system tables are created with the proper access rights when you log in to your database for the first time
- Display and open a PowerBuilder extended attribute system table
- Understand the kind of information stored in the PowerBuilder extended attribute system tables
- Control extended attribute system table access

Logging in to your database for the first time

By default, PowerBuilder creates the extended attribute system tables the first time you connect to a database.

To ensure that PowerBuilder creates the extended attribute system tables with the proper access rights to make them available to all users, the first person to connect to the database with PowerBuilder must log in with the proper authority.

❖ To ensure proper creation of the PowerBuilder extended attribute system tables:

- Make sure the first person to connect to the database with PowerBuilder has sufficient authority to create tables and grant permissions to PUBLIC.

  This means that the first person to connect to the database should log in as the database owner, database administrator, system user, system administrator, or system owner, as specified by your DBMS.

Creating the extended attribute system tables when using the DirectConnect interface

When you are using the DirectConnect interface, the PowerBuilder extended attribute system tables are *not* created automatically the first time you connect to a database. You must run the *DB2SYSPB.SQL* script to create the system tables, as described in Using the DB2SYSPB.SQL script on page 163.

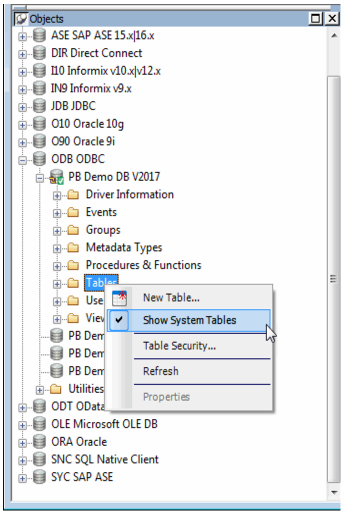
Displaying the PowerBuilder extended attribute system tables

PowerBuilder updates the extended attribute system tables automatically whenever you change the information for a table or column. The PowerBuilder extended attribute system tables are different from the system tables provided by your DBMS.

You can display and open PowerBuilder extended attribute system tables in the Database painter just like other tables.

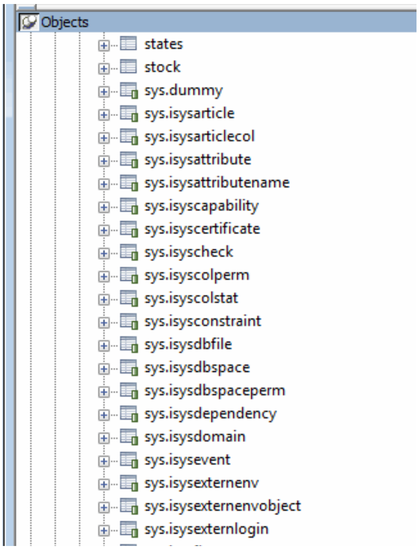
❖ **To display the PowerBuilder extended attribute system tables:**

1 In the Database painter, highlight Tables in the list of database objects for the active connection and select Show System Tables from the pop-up menu.



2 The PowerBuilder extended attribute system tables and DBMS system tables display in the tables list, as follows:

- **PowerBuilder system tables** The five system tables are: pbcatcol, pbcatedt, pbcatfmt, pbcattbl, and pbcatvld.
- **DBMS system tables** The system tables supplied by the DBMS usually have a DBMS-specific prefix (such as *sys* or *dbo*).



Objects
states
stock
sys.dummy
sys.isysarticle
sys.isysarticlecol
sys.isysattribute
sys.isysattributename
sys.isyscapability
sys.isyscertificate
sys.isyscheck
sys.isyscolperm
sys.isyscolstat
sys.isysconstraint
sys.isysdbfile
sys.isysdbspace
sys.isysdbspaceperm
sys.isysdependency
sys.isysdomain
sys.isysevent
sys.isysexternenv
sys.isysexternenvobject
sys.isysexternlogin

3 Display the contents of a PowerBuilder system table in the Object Layout, Object Details, and/or Columns views.

For instructions, see the *Users Guide*.

Do not edit the extended attribute system tables
Do not change the values in the PowerBuilder extended attribute system tables.

Contents of the extended attribute system tables

PowerBuilder stores five types of extended attribute information in the system tables as described in Table 13-1.

Table 13-1: Extended attribute system tables

System table	Information about	Attributes
pbcatcol	Columns	Names, comments, headers, labels, case, initial value, and justification
pbcatedt	Edit styles	Edit style names and definitions
pbcatfmt	Display formats	Display format names and definitions
pbcattbl	Tables	Name, owner, default fonts (for data, headings and labels), and comments
pbcatvld	Validation rules	Validation rule names and definitions

For more about the PowerBuilder system tables, see the Appendix in the *Users Guide*.

Prefixes in system table names
For some databases, PowerBuilder precedes the name of the system table with a default DBMS-specific prefix. For example, the names of PowerBuilder system tables have the prefix DBO in a SQL Server database (such as DBO.pbcatcol), or SYSTEM in an Oracle database (such as SYSTEM.pbcatfmt).

The preceding table gives the base name of each system table without the DBMS-specific prefix.

Controlling system table access

To control access to the PowerBuilder system tables at your site, you can specify that PowerBuilder not create or update the system tables or that the system tables be accessible only to certain users or groups.

You can control system table access by doing any of the following:

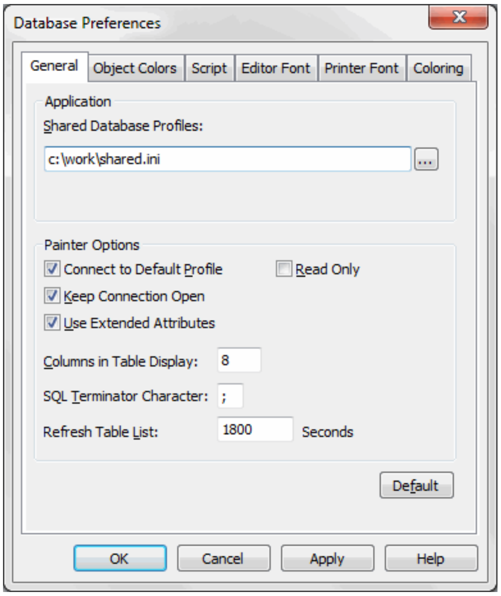
- **Setting Use Extended Attributes** Set the Use Extended Attributes database preference in the Database Preferences dialog box in the Database painter.

- **Setting Read Only** Set the Read Only database preference in the Database Preferences dialog box in the Database painter.
- **Granting permissions on the system tables** Grant explicit permissions on the system tables to users or groups at your site.

Setting Use Extended Attributes or Read Only to control access

❖ To control system table access by setting Use Extended Attributes or Read Only:

1 Select Design>Options from the menu bar to display the Database Preferences dialog box.



Shared Database Profiles: c:\work\shared.ini

☑ Connect to Default Profile
☐ Read Only
☑ Keep Connection Open
☑ Use Extended Attributes

Columns in Table Display: 8

SQL Terminator Character: ;

Refresh Table List: 1800 Seconds

2 On the General page, set values for Use Extended Attributes or Read Only as follows:

Preference	What you do	Effect
Use Extended Attributes	Clear the check box	Does not create the PowerBuilder system tables if they do not exist. Instead, the painter uses the appropriate default values for extended attributes (such as headers, labels, and text color). If the PowerBuilder system tables already exist, PowerBuilder does not use them when you create a new report.
Read Only	Select the check box	If the PowerBuilder system tables already exist, PowerBuilder uses them when you create a new report, *but does not update them*. You *cannot* modify (update) information in the system tables or any other database tables in the Report painter when the Read Only check box is selected.

3 Click OK.

PowerBuilder applies the preference settings to the current connection and all future connections and saves them in the registry.

Granting permissions on system tables to control access

If your DBMS supports SQL GRANT and REVOKE statements, you can control access to the PowerBuilder system tables. The default authorization for each repository table is:

GRANT SELECT, UPDATE, INSERT, DELETE ON *table* TO PUBLIC

After the system tables are created, you can (for example) control access to them by granting SELECT authority to end users and SELECT, UPDATE, INSERT, and DELETE authority to developers. This technique offers security and flexibility that is enforced by the DBMS itself.

CHAPTER 14 Setting Additional Connection Parameters

About this chapter

To fine-tune your database connection and take advantage of DBMS-specific features that your interface supports, you can set additional connection parameters at any time. These additional connection parameters include:

- Database parameters
- Database preferences

These connection parameters are described in the Database Connectivity section in the online Help.

This chapter describes how to set database parameters and database preferences in PowerBuilder.

Contents

Basic steps for setting connection parameters

This section gives basic steps for setting database parameters and database preferences in PowerBuilder.

❖ **To set database parameters:**

1 Learn how to set database parameters in the development environment or in code.

See Setting database parameters on page 187.

2. Determine the database parameters you can set for your database interface.

   For a table listing each supported database interface and the database parameters you can use with that interface, see "Database parameters and supported database interfaces" in the online Help.

3. Read the description of the database parameter you want to set in the online Help.

4. Set the database parameter for your database connection.

❖ **To set database preferences:**

1. Learn how to set database preferences in the development environment or PowerBuilder application script.

   See Setting database preferences on page 190.

2. Determine the database preferences you can set for your DBMS.

   For a table listing each supported database interface and the database preferences you can use with that interface, see "Database parameters and supported database interfaces" in the online Help.

3. Read the description of the database preference you want to set in the online Help.

4. Set the database preference for your database connection.

About the Database Profile Setup dialog box

The interface-specific Database Profile Setup dialog box makes it easy to set additional connection parameters in the development environment or in code. You can:

- Supply values for connection options supported by your database interface

  Each database interface has its own Database Profile Setup dialog box that includes settings only for those connection parameters supported by the interface. Similar parameters are grouped on the same tab page. The Database Profile Setup dialog box for *all* interfaces includes the Connection tab and Preview tab. Depending on the requirements and features of your interface, one or more other tab pages might also display.

- Easily set additional connection parameters in the development environment

  You can specify additional connection parameters (database parameters and transaction object properties) with easy-to-use check boxes, drop-down lists, and text boxes. PowerBuilder generates the proper syntax automatically when it saves your database profile in the system registry.

- Generate connection syntax for use in your PowerBuilder application script

  As you complete the Database Profile Setup dialog box in PowerBuilder, the correct connection syntax for each selected option is generated on the Preview tab. PowerBuilder assigns the corresponding database parameter or transaction object property name to each option and inserts quotation marks, commas, semicolons, and other characters where needed. You can copy the syntax you want from the Preview tab into your PowerBuilder script.

Setting database parameters

In PowerBuilder, you can set database parameters by doing either of the following:

- Editing the Database Profile Setup dialog box for your connection in the development environment
- Specifying connection parameters in an application script

Setting database parameters in the development environment

Editing database profiles

To set database parameters for a database connection in the PowerBuilder development environment, you must edit the database profile for that connection.

Character limit for strings

Strings containing database parameters that you specify in the Database Profile Setup dialog box for your connection can be up to 999 characters in length.

This limit applies only to database parameters that you set in a database profile in the development environment. Database strings specified in code as properties of the Transaction object are *not* limited to a specified length.

Setting database parameters in a PowerBuilder application script

If you are developing an application that connects to a database, you must specify the required connection parameters in the appropriate script as properties of the default Transaction object (SQLCA) or a Transaction object that you create. For example, you might specify connection parameters in the script that opens the application.

One of the connection parameters you might want to specify in a script is DBParm. You can do this by:

- *(Recommended)* Copying DBParm syntax from the Preview tab in the Database Profile Setup dialog box into your script
- Coding PowerScript to set values for the DBParm property of the Transaction object
- Reading DBParm values from an external text file

Copying DBParm syntax from the Preview tab

The easiest way to specify DBParm parameters in a PowerBuilder application script is to copy the DBParm syntax from the Preview tab in the Database Profile Setup dialog box into your code, modifying the default Transaction object name (SQLCA) if necessary.

As you set parameters in the Database Profile Setup dialog box in the development environment, PowerBuilder generates the correct connection syntax on the Preview tab. Therefore, copying the syntax directly from the Preview tab ensures that you use the correct DBParm syntax in your code.

❖ **To copy DBParm syntax from the Preview tab into your code:**

1. On one or more tab pages in the Database Profile Setup dialog box for your connection, supply values for any parameters you want to set.

   For instructions, see Setting database parameters in the development environment on page 187.

   For information about the parameters for your interface and the values to supply, click Help.

2. Click Apply to save your changes to the current tab without closing the Database Profile Setup dialog box.

3. Click the Preview tab.

   The correct DBParm syntax for each selected option displays in the Database Connection Syntax box.

4 Select one or more lines of text in the Database Connection Syntax box and click Copy.

PowerBuilder copies the selected text to the clipboard.

5 Click OK to close the Database Profile Setup dialog box.

6 Paste the selected text from the Preview tab into your code, modifying the default Transaction object name (SQLCA) if necessary.

Coding PowerScript to set values for the DBParm property

Another way to specify connection parameters in a script is by coding PowerScript to assign values to properties of the Transaction object. PowerBuilder uses a special nonvisual object called a **Transaction object** to communicate with the database. The default Transaction object is named SQLCA, which stands for SQL Communications Area.

SQLCA has 15 properties, 10 of which are used to connect to your database. One of the 10 connection properties is DBParm. DBParm contains DBMS-specific parameters that let your application take advantage of various features supported by the database interface.

❖ To set values for the DBParm property in a PowerBuilder script:

1 Open the application script in which you want to specify connection parameters.

For instructions, see the *Users Guide*.

2 Use the following PowerScript syntax to specify DBParm parameters. Make sure you separate the DBParm parameters with commas, and enclose the entire DBParm string in double quotes.

SQLCA.dbParm = "*parameter_1*, *parameter_2*, *parameter_n*"

For example, the following statement in a PowerBuilder script sets the DBParm property for an ODBC data source named Sales. In this example, the DBParm property consists of two parameters: ConnectString and Async.

```
SQLCA.dbParm="ConnectString='DSN=Sales;UID=PB;
   PWD=xyz',Async=1"
```

3 Compile the PowerBuilder script to save your changes.

For instructions, see the *Users Guide*.

Reading DBParm values from an external text file

As an alternative to setting the DBParm property in a PowerBuilder application script, you can use the PowerScript ProfileString function to read DBParm values from a specified section of an external text file, such as an application-specific initialization file.

❖ **To read DBParm values from an external text file:**

1. Open the application script in which you want to specify connection parameters.

   For instructions, see the *Users Guide*.

2. Use the following PowerScript syntax to specify the ProfileString function with the SQLCA.DBParm property:

   SQLCA.dbParm = **ProfileString** (*file*, *section*, *key*,
   default)

   For example, the following statement in a PowerBuilder script reads the DBParm values from the [Database] section of the *APP.INI* file:

   ```
   SQLCA.dbParm=ProfileString("APP.INI","Database",
      "dbParm","")
   ```

3. Compile the script to save your changes.

   For instructions, see the *Users Guide*.

Setting database preferences

How to set

The way you set connection-related database preferences in PowerBuilder varies, as summarized in the following table (AutoCommit and Lock are the only database preferences that you can set in a PowerBuilder application script).

Table 14-1: Database preferences and where they can be set

Database preference	Set in development environment by editing	Set in PowerBuilder application by editing
AutoCommit	Database Profile Setup dialog box for your connection	Application script
Lock	Database Profile Setup dialog box for your connection	Application script

Database preference	Set in development environment by editing	Set in PowerBuilder application by editing
Shared Database Profiles	Database Preferences property sheet	—
Connect to Default Profile	Database Preferences property sheet	—
Read Only	Database Preferences property sheet	—
Keep Connection Open	Database Preferences property sheet	—
Use Extended Attributes	Database Preferences property sheet	—
SQL Terminator Character	Database Preferences property sheet	—

The following sections give the steps for setting database preferences in the development environment and (for AutoCommit and Lock) in a PowerBuilder application script.

For more information For information about using a specific database preference, see its description in the online Help.

Setting database preferences in the development environment

There are two ways to set database preferences in the PowerBuilder development environment on *all* supported development platforms, depending on the preference you want to set:

- Set AutoCommit and Lock (Isolation Level) in the Database Profile Setup dialog box for your connection

  ADO.NET
  For ADO.NET, Isolation is a database parameter.

- Set all other database preferences in the Database Preferences dialog box in the Database painter

Setting AutoCommit and Lock in the database profile

The AutoCommit and Lock (Isolation Level) preferences are properties of the default Transaction object, SQLCA. For AutoCommit and Lock to take effect in the PowerBuilder development environment, you must specify them *before* you connect to a database. Changes to these preferences after the connection occurs have no effect on the current connection.

To set AutoCommit and Lock before PowerBuilder connects to your database, you specify their values in the Database Profile Setup dialog box for your connection.

❖ To set AutoCommit and Lock (Isolation Level) in a database profile:

1. Display the Database Profiles dialog box.
2. Click the plus sign (+) to the left of the interface you are using or double-click the interface name.

   The list expands to display the database profiles defined for your interface.
3. Select the name of the profile you want and click Edit.

   The Database Profile Setup dialog box for the selected profile displays.
4. On the Connection tab page, supply values for one or both of the following:

   - **Isolation Level** If your database supports the use of locking and isolation levels, select the isolation level you want to use for this connection from the Isolation Level drop-down list. (The Isolation Level drop-down list contains valid lock values for your interface.)
   - **AutoCommit Mode** The setting of AutoCommit controls whether PowerBuilder issues SQL statements outside (True) or inside (False) the scope of a transaction. *If your database supports it*, select the AutoCommit Mode check box to set AutoCommit to True or clear the AutoCommit Mode check box (the default) to set AutoCommit to False.

   For example, in addition to values for basic connection parameters (Server, Login ID, Password, and Database), the Connection tab page for the following SAP Adaptive Server Enterprise profile named Sales shows nondefault settings for Isolation Level and AutoCommit Mode.

5 (Optional) In PowerBuilder, click the Preview tab if you want to see the PowerScript connection syntax generated for Lock and AutoCommit.

PowerBuilder generates correct PowerScript connection syntax for each option you set in the Database Profile Setup dialog box. You can copy this syntax directly into a PowerBuilder application script.

For instructions, see Copying DBParm syntax from the Preview tab on page 188.

6 Click OK to close the Database Profile Setup dialog box.

PowerBuilder saves your settings in the database profile entry in the registry.

Setting preferences in the Database Preferences dialog box

To set the following connection-related database preferences, complete the Database Preferences dialog box in the PowerBuilder Database painter:

- Shared Database Profiles
- Connect to Default Profile
- Read Only
- Keep Connection Open
- Use Extended Attributes
- SQL Terminator Character

Other database preferences
The Database Preferences dialog box also lets you set other database preferences that affect the behavior of the Database painter itself. For information about the other preferences you can set in the Database Preferences dialog box, see the *Users Guide*.

❖ To set connection-related preferences in the Database Preferences dialog box:

1 Open the Database painter.

2 Select Design>Options from the menu bar.

The Database Preferences dialog box displays. If necessary, click the General tab to display the General property page.

3 Specify values for one or more of the connection-related database preferences in the following table.

Table 14-2: Connection-related database preferences

Preference	Description	For details, see
Shared Database Profiles	Specifies the pathname of the file containing the database profiles you want to share. You can type the pathname or click Browse to display it.	Sharing database profiles on page 173
Connect to Default Profile	Controls whether the Database painter establishes a connection to a database using a default profile when the painter is invoked. If not selected, the Database painter opens without establishing a connection to a database.	Connect to Default Profile in online Help
Read Only	Specifies whether PowerBuilder should update the extended attribute system tables and any other tables in your database. Select or clear the Read Only check box as follows: • **Select the check box** Does not update the extended attribute system tables or any other tables in your database. You *cannot* modify (update) information in the extended attribute system tables or any other database tables from the DataWindow painter when the Read Only check box is selected. • **Clear the check box** (Default) Updates the extended attribute system tables and any other tables in your database.	Read Only in the online Help
Keep Connection Open	When you connect to a database in PowerBuilder without using a database profile, specifies when PowerBuilder closes the connection. Select or clear the Keep Connection Open check box as follows: • **Select the check box** (Default) Stays connected to the database throughout your session and closes the connection when you exit • **Clear the check box** Opens the connection only when a painter requests it and closes the connection when you close a painter or finish compiling a script **Not used with profile** This preference has no effect when you connect using a database profile.	Keep Connection Open in the online Help

Preference	Description	For details, see
Use Extended Attributes	Specifies whether PowerBuilder should create and use the extended attribute system tables. Select or clear the Use Extended Attributes check box as follows: • **Select the check box** (Default) Creates and uses the extended attribute system tables • **Clear the check box** Does *not* create the extended attribute system tables	Use Extended Attributes in the online Help
Columns in Table Display	Specify the number of table columns to be displayed when InfoMaker displays a table graphically. The default is eight.	

4 Do one of the following:

- Click Apply to apply the preference settings to the current connection without closing the Database Preferences dialog box.
- Click OK to apply the preference settings to the current connection and close the Database Preferences dialog box.

PowerBuilder saves your preference settings in the database section of *PB.INI*.

Setting AutoCommit and Lock in a PowerBuilder application script

If you are developing a PowerBuilder application that connects to a database, you must specify the required connection parameters in the appropriate script as properties of the default Transaction object (SQLCA) or a Transaction object that you create. For example, you might specify connection parameters in the script that opens the application.

AutoCommit and Lock are properties of SQLCA. As such, they are the *only* database preferences you can set in a PowerBuilder script. You can do this by:

- *(Recommended)* Copying PowerScript syntax for AutoCommit and Lock from the Preview tab in the Database Profile Setup dialog box into your script
- Coding PowerScript to set values for the AutoCommit and Lock properties of the Transaction object
- Reading AutoCommit and Lock values from an external text file

For more about using Transaction objects to communicate with a database in a PowerBuilder application, see *Application Techniques*.

Copying AutoCommit and Lock syntax from the Preview tab

The easiest way to specify AutoCommit and Lock in a PowerBuilder application script is to copy the PowerScript syntax from the Preview tab in the Database Profile Setup dialog box into your script, modifying the default Transaction object name (SQLCA) if necessary.

As you complete the Database Profile Setup dialog box in the development environment, PowerBuilder generates the correct connection syntax on the Preview tab for each selected option. Therefore, copying the syntax directly from the Preview tab ensures that you use the correct PowerScript syntax in your script.

❖ To copy AutoCommit and Lock syntax from the Preview tab into your script:

1 On the Connection tab in the Database Profile Setup dialog box for your connection, supply values for AutoCommit and Lock (Isolation Level) as required.

For instructions, see Setting AutoCommit and Lock in a PowerBuilder application script on page 195.

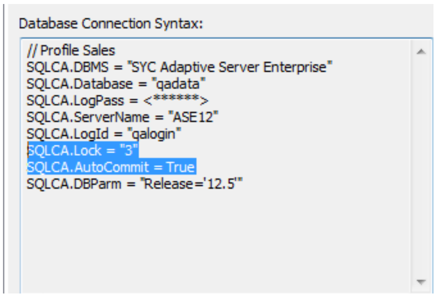
For example, in addition to values for basic connection parameters (Server, Login ID, Password, and Database), the Connection tab for the following Adaptive Server profile named Sales shows nondefault settings for Isolation Level and AutoCommit Mode.

For information about the DBParm parameters for your interface and the values to supply, click Help.

2 Click Apply to save your changes to the current tab without closing the Database Profile Setup dialog box.

3 Click the Preview tab.

The correct PowerScript syntax for each selected option displays in the Database Connection Syntax box. For example:



```
// Profile Sales
SQLCA.DBMS = "SYC Adaptive Server Enterprise"
SQLCA.Database = "qadata"
SQLCA.LogPass = <******>
SQLCA.ServerName = "ASE12"
SQLCA.LogId = "qalogin"
SQLCA.Lock = "3"
SQLCA.AutoCommit = True
SQLCA.DBParm = "Release='12.5'"
```

4 Select one or more lines of text in the Database Connection Syntax box and click Copy.

PowerBuilder copies the selected text to the clipboard.

5 Click OK to close the Database Profile Setup dialog box.

6 Paste the selected text from the Preview tab into your script, modifying the default Transaction object name (SQLCA) if necessary.

Coding PowerScript to set values for AutoCommit and Lock

Another way to specify the AutoCommit and Lock properties in a script is by coding PowerScript to assign values to the AutoCommit and Lock properties of the Transaction object. PowerBuilder uses a special nongraphic object called a **Transaction object** to communicate with the database. The default Transaction object is named SQLCA, which stands for SQL Communications Area.

SQLCA has 15 properties, 10 of which are used to connect to your database. Two of the connection properties are AutoCommit and Lock, which you can set as described in the following procedure.

❖ To set the AutoCommit and Lock properties in a PowerBuilder script:

1 Open the application script in which you want to set connection properties.

For instructions, see the *Users Guide*.

2 Use the following PowerScript syntax to set the AutoCommit and Lock properties. (This syntax assumes you are using the default Transaction object SQLCA, but you can also define your own Transaction object.)

SQLCA.AutoCommit = *value*

SQLCA.Lock = "*value*"

For example, the following statements in a PowerBuilder script use the default Transaction object SQLCA to connect to a SAP Adaptive Server Enterprise database named Test. SQLCA.AutoCommit is set to True and SQLCA.Lock is set to isolation level 3 (Serializable transactions).

```
SQLCA.DBMS        = "SYC"
SQLCA.Database    = "Test"
SQLCA.LogID       = "Frans"
SQLCA.LogPass     = "xxyyzz"
SQLCA.ServerName  = "HOST1"
SQLCA.AutoCommit  = True
SQLCA.Lock        = "3"
```

For more information, see AutoCommit or Lock in the online Help.

3 Compile the script to save your changes.

For instructions, see the *Users Guide*.

Reading AutoCommit and Lock values from an external text file

As an alternative to setting the AutoCommit and Lock properties in a PowerBuilder application script, you can use the PowerScript ProfileString function to read the AutoCommit and Lock values from a specified section of an external text file, such as an application-specific initialization file.

❖ **To read AutoCommit and Lock values from an external text file:**

1 Open the application script in which you want to set connection properties.

2 Use the following PowerScript syntax to specify the ProfileString function with the SQLCA.Lock property:

SQLCA.Lock = **ProfileString** (*file*, *section*, *key*, *default*)

The AutoCommit property is a boolean, so you need to convert the string returned by ProfileString to a boolean. For example, the following statements in a PowerBuilder script read the AutoCommit and Lock values from the [Database] section of the *APP.INI* file:

```
string ls_string
ls_string=Upper(ProfileString("APP.INI","Database",
   "Autocommit",""))
if ls_string = "TRUE" then
   SQLCA.Autocommit = TRUE
else
   SQLCA.Autocommit = FALSE
end if
SQLCA.Lock=ProfileString("APP.INI","Database",
   "Lock","")
```

3 Compile the script to save your changes.

Getting values from the registry

If the AutoCommit and Lock values are stored in an application settings key in the registry, use the RegistryGet function to obtain them. For example:

```
string ls_string
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp", &
   "Autocommit", RegString!, ls_string)
```

```
if Upper(ls_string) = "TRUE" then
   SQLCA.Autocommit = TRUE
else
   SQLCA.Autocommit = FALSE
end if
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp", &
   "Lock", RegString!, ls_string)
```

PART 5

Working with Transaction Servers

This part describes how to make database connections for transactional components.

CHAPTER 15 Troubleshooting Your Connection

About this chapter

This chapter describes how to troubleshoot your database connection in PowerBuilder by using the following tools:

- Database Trace
- SQL Statement Trace
- ODBC Driver Manager Trace
- JDBC Driver Manager Trace

Contents

Overview of troubleshooting tools

When you use PowerBuilder, there are several tools available to trace your database connection in order to troubleshoot problems.

Table 15-1: Database trace tools

Use this tool	To trace a connection to
Database Trace	*Any* database that PowerBuilder accesses through one of the database interfaces
ODBC Driver Manager Trace	An ODBC data source only
JDBC Driver Manager Trace	A JDBC database only

Using the Database Trace tool

This section describes how to use the Database Trace tool.

About the Database Trace tool

The Database Trace tool records the internal commands that PowerBuilder executes while accessing a database. You can trace a database connection in the development environment or in a PowerBuilder application that connects to a database.

PowerBuilder writes the output of Database Trace to a log file named *DBTRACE.LOG* (by default) or to a nondefault log file that you specify. When you enable database tracing for the first time, PowerBuilder creates the log file on your computer. Tracing continues until you disconnect from the database.

Using the Database Trace tool with one connection
You can use the Database Trace tool for only one DBMS at a time and for one database connection at a time.

For example, if your application connects to both an ODBC data source and an Adaptive Server Enterprise database, you can trace either the ODBC connection or the Adaptive Server Enterprise connection, but not both connections at the same time.

How you can use the Database Trace tool

You can use information from the Database Trace tool to understand what PowerBuilder is doing *internally* when you work with your database. Examining the information in the log file can help you:

- Understand how PowerBuilder interacts with your database
- Identify and resolve problems with your database connection
- Provide useful information to Technical Support if you call them for help with your database connection

If you are familiar with PowerBuilder and your DBMS, you can use the information in the log to help troubleshoot connection problems on your own. If you are less experienced or need help, run the Database Trace tool *before* you call Technical Support. You can then report or send the results of the trace to the Technical Support representative who takes your call.

Contents of the Database Trace log

Default contents of the trace file

By default, the Database Trace tool records the following information in the log file when you trace a database connection:

- Parameters used to connect to the database
- Time to perform each database operation (in microseconds)
- The internal commands executed to retrieve and display table and column information from your database. Examples include:
  - Preparing and executing SQL statements such as SELECT, INSERT, UPDATE, and DELETE
  - Getting column descriptions
  - Fetching table rows
  - Binding user-supplied values to columns (if your database supports bind variables)
  - Committing and rolling back database changes
- Disconnecting from the database
- Shutting down the database interface

You can opt to include the names of DBI commands and the time elapsed from the last database connection to the completion of processing for each log entry. You can exclude binding and timing information as well as the data from all fetch requests.

Database Trace dialog box selections

The Database Trace dialog box lets you select the following items for inclusion in or exclusion from a database trace file:

- **Bind variables** Metadata about the result set columns obtained from the database
- **Fetch buffers** Data values returned from each fetch request
- **DBI names** Database interface commands that are processed
- **Time to implement request** Time required to process DBI commands; the interval is measured in thousandths of milliseconds (microseconds)
- **Cumulative time** Cumulative total of timings since the database connection began; the timing measurement is in thousandths of milliseconds

Registry settings for DBTrace

The selections made in the Database Trace dialog box are saved to the registry of the machine from which the database connections are made. Windows registry settings for the database trace utility configuration are stored under the *HKEY_CURRENT_USER\Software\Sybase\PowerBuilder\17.0\DBTrace* key. Registry strings under this key are: ShowBindings, FetchBuffers, ShowDBINames, Timing, SumTiming, LogFileName, and ShowDialog. Except for the LogFileName string to which you can assign a full file name for the trace output file, all strings can be set to either 0 or 1.

The ShowDialog registry string can be set to prevent display of the Database Trace dialog box when a database connection is made with tracing enabled. This is the only one of the trace registry strings that you cannot change from the Database Trace dialog box. You must set ShowDialog to 0 in the registry to keep the configuration dialog box from displaying.

INI file settings for DBTrace

If you do not have access to the registry, you can use *PB.INI* to store trace file settings. Add a [DbTrace] section to the INI file with at least one of the following values set, then restart PowerBuilder:

```
[DbTrace]
ShowDBINames=0
FetchBuffers=1
ShowBindings=1
SumTiming=1
Timing=1
ShowDialog=1
LogFileName=dbtrace.log
```

The keywords are the same as in the registry and have the same meaning. When you connect to the database again, the initial settings are taken from the INI file, and when you modify them, the changes are written to the INI file.

If the file name for LogFileName does not include an absolute path, the log file is written to the following path, where *<username>* is your login ID: *Users\<username>\AppData\Local\Appeon\PowerBuilder 17.0*. If there are no DbTrace settings in the INI file, the registry settings are used.

Error messages

If the database trace utility cannot open the trace output file with write access, an error message lets you know that the specified trace file could not be created or opened. If the trace utility driver cannot be loaded successfully, a message box informs you that the selected Trace DBMS is not supported in your current installation.

Format of the Database Trace log

The specific content of the Database Trace log file depends on the database you are accessing and the operations you are performing. However, the log uses the following basic format to display output:

COMMAND: (*time*)

{*additional_information*}

Parameter	Description
COMMAND	The internal command that PowerBuilder executes to perform the database operation.
time	The number of microseconds it takes PowerBuilder to perform the database operation. The precision used depends on your operating system's timing mechanism.
additional_information	(Optional) Additional information about the command. The information provided depends on the database operation.

Example

The following portion of the log file shows the commands PowerBuilder executes to fetch two rows from a SQL Anywhere database table:

```
FETCH NEXT: (0.479 MS)
 COLUMN=400 COLUMN=Marketing COLUMN=Evans
FETCH NEXT: (0.001 MS)
 COLUMN=500 COLUMN=Shipping COLUMN=Martinez
```

If you opt to include DBI Names and Sum Time information in the trace log file, the log for the same two rows might look like this:

```
FETCH NEXT:(DBI_FETCHNEXT) (1.459 MS / 3858.556 MS)
 COLUMN=400 COLUMN=Marketing COLUMN=Evans
FETCH NEXT:(DBI_FETCHNEXT) (0.001 MS / 3858.557 MS)
 COLUMN=500 COLUMN=Shipping COLUMN=Martinez
```

For a more complete example of Database Trace output, see Sample Database Trace output on page 215.

Starting the Database Trace tool

By default, the Database Trace tool is turned off in PowerBuilder. You can start it in the PowerBuilder development environment or in a PowerBuilder application to trace your database connection.

Turning tracing on and off
To turn tracing on or off you must reconnect. Setting and resetting are not sufficient.

Starting Database Trace in the development environment

To start the Database Trace tool in the PowerBuilder development environment, edit the database profile for the connection you want to trace, as described in the following procedure.

❖ **To start the Database Trace tool by editing a database profile:**

1 Open the Database Profile Setup dialog box for the connection you want to trace.

2 On the Connection tab, select the Generate Trace check box and click OK or Apply. (The Generate Trace check box is located on the System tab in the OLE DB Database Profile Setup dialog box.)

  The Database Profiles dialog box displays with the name of the edited profile highlighted.

  For example, here is the relevant portion of a database profile entry for Adaptive Server 12.5 Test. The setting that starts Database Trace is DBMS:

```
[Default]      [value not set]
AutoCommit     "FALSE"
Database       "qadata"
DatabasePassword  "00"
DBMS           "TRACE SYC Adaptive Server Enterprise"
DbParm         "Release='12.5'"
Lock           ""
LogId          "qalogin"
LogPassword    "00171717171717"
Prompt         "FALSE"
ServerName     "Host125"
UserID         ""
```

3 Click Connect in the Database Profiles dialog box to connect to the database.

The Database Trace dialog box displays, indicating that database tracing is enabled. You can enter the file location where PowerBuilder writes the trace output. By default, PowerBuilder writes Database Trace output to a log file named *DBTRACE.LOG*. You can change the log file name and location in the Database Trace dialog box.

The Database Trace dialog box also lets you select the level of tracing information that you want in the database trace file.

4 Select the types of items you want to include in the trace file and click OK.

PowerBuilder connects to the database and starts tracing the connection.

Starting Database Trace in a PowerBuilder application

In a PowerBuilder application that connects to a database, you must specify the required connection parameters in the appropriate script. For example, you might specify them in the script that opens the application.

To trace a database connection in a PowerBuilder script, you specify the name of the DBMS preceded by the word *trace* and a single space. You can do this by:

- Copying the PowerScript DBMS trace syntax from the Preview tab in the Database Profile Setup dialog box into your script
- Coding PowerScript to set a value for the DBMS property of the Transaction object
- Reading the DBMS value from an external text file

For more about using Transaction objects to communicate with a database in a PowerBuilder application, see *Application Techniques*.

Copying DBMS trace syntax from the Preview tab

One way to start Database Trace in a PowerBuilder application script is to copy the PowerScript DBMS trace syntax from the Preview tab in the Database Profile Setup dialog box into your script, modifying the default Transaction object name (SQLCA) if necessary.

As you complete the Database Profile Setup dialog box in the development environment, PowerBuilder generates the correct connection syntax on the Preview tab for each selected option, including Generate Trace. Therefore, copying the syntax directly from the Preview tab ensures that it is accurate in your script.

❖ **To copy DBMS trace syntax from the Preview tab into your script:**

1. On the Connection tab (or System tab in the case of OLE DB) in the Database Profile Setup dialog box for your connection, select the Generate Trace check box to turn on Database Trace.

   For instructions, see Starting Database Trace in the development environment on page 208.

2. Click Apply to save your changes to the Connection tab without closing the Database Profile Setup dialog box.

3. Click the Preview tab.

   The correct PowerScript connection syntax for the Generate Trace and other selected options displays in the Database Connection Syntax box.

4. Select the SQLCA.DBMS line and any other syntax you want to copy to your script and click Copy.

   PowerBuilder copies the selected text to the clipboard.

5. Click OK to close the Database Profile Setup dialog box.

6. Paste the selected text from the Preview tab into your script, modifying the default Transaction object name (SQLCA) if necessary.

Coding PowerScript to set a value for the DBMS property

Another way to start the Database Trace tool in a PowerBuilder script is to specify it as part of the DBMS property of the Transaction object. The **Transaction object** is a special nonvisual object that PowerBuilder uses to communicate with the database. The default Transaction object is named SQLCA, which stands for SQL Communications Area.

SQLCA has 15 properties, 10 of which are used to connect to your database. One of the 10 connection properties is DBMS. The DBMS property contains the name of the database to which you want to connect.

❖ **To start the Database Trace tool by specifying the DBMS property:**

- Use the following PowerScript syntax to specify the DBMS property. (This syntax assumes you are using the default Transaction object SQLCA, but you can also define your own Transaction object.)

  SQLCA.DBMS = "trace *DBMS_name*"

  For example, the following statements in a PowerBuilder script set the SQLCA properties required to connect to an Adaptive Server database named Test. The keyword *trace* in the DBMS property indicates that you want to trace the database connection.

```
SQLCA.DBMS          = "trace SYC"
SQLCA.database      = "Test"
SQLCA.logId         = "Frans"
SQLCA.LogPass       = "xxyyzz"
SQLCA.ServerName    = "Tomlin"
```

Reading the DBMS value from an external text file or the registry

As an alternative to setting the DBMS property in your PowerBuilder application script, you can use the PowerScript ProfileString function to read the DBMS value from a specified section of an external text file, such as an application-specific initialization file, or from an application settings key in the registry.

The following procedure assumes that the DBMS value read from the database section in your initialization file uses the following syntax to enable database tracing:

DBMS = trace *DBMS_name*

❖ **To start the Database Trace tool by reading the DBMS value from an external text file:**

- Use the following PowerScript syntax to specify the ProfileString function with the DBMS property:

  SQLCA.DBMS = ProfileString(*file*, *section*, *variable*, *default_value*)

  For example, the following statement in a PowerBuilder script reads the DBMS value from the [Database] section of the *APP.INI* file:

  ```
  SQLCA.DBMS=ProfileString("APP.INI","Database",
     "DBMS","")
  ```

For how to get a value from a registry file instead, see Getting values from the registry on page 198.

Starting a trace in PowerScript with the PBTrace parameter

Instead of tracing all database commands from the start of a database connection, you can start and end a trace programmatically for specific database queries. To start a trace, you can assign the string value pair “PBTrace=1” to the transaction object DBParm property; to end a trace, you assign the string value pair “PBTrace=0”.

For example, if you wanted data to be logged to the trace output for a single retrieve command, you could disable tracing from the start of the connection and then surround the retrieve call with DBParm property assignments as follows:

```
SQLCA.DBMS = "TRACE ODBC"
SQLCA.DBParm="PBTrace=0"
Connect using SQLCA;
...
SQLCA.DBParm="PBTrace=1"
dw_1.Retrieve ( )
SQLCA.DBParm="PBTrace=0"
```

When you first connect to a database after setting the DBMS parameter to "Trace *DBMSName*", a configuration dialog box displays. The configuration parameters that you set in this dialog box are saved to the registry. Configuration parameters are retrieved from the registry when you begin tracing by assigning the DBParm parameter to "PBTrace=1".

You can start and stop the SQL statement trace utility in the same way if you set the DBMS value to "TRS *DBMSName*" instead of "Trace *DBMSName*". For information about the SQL statement trace utility, see Using the SQL statement trace utility on page 217.

Stopping the Database Trace tool

Once you start tracing a particular database connection, PowerBuilder continues sending trace output to the log until you do one of the following:

- Reconnect to the same database with tracing stopped
- Connect to another database for which you have not enabled tracing

Stopping Database Trace in the development environment

❖ **To stop the Database Trace tool by editing a database profile:**

1. In the Database Profile Setup dialog box for the database you are tracing, clear the Generate Trace check box on the Connection tab.
2. Click OK in the Database Profile Setup dialog box.

   The Database Profiles dialog box displays with the name of the edited profile highlighted.

3 Right-click on the connected database and select Re-connect from the drop-down menu in the Database Profiles dialog box.

PowerBuilder connects to the database and stops tracing the connection.

Stopping Database Trace in a PowerBuilder application

To stop Database Trace in a PowerBuilder application script, you must delete the word *trace* from the DBMS property. You can do this by:

- Editing the value of the DBMS property of the Transaction object
- Reading the DBMS value from an external text file

You must reconnect for the change to take effect.

Editing the DBMS property

❖ **To stop Database Trace by editing the DBMS value in a PowerBuilder script:**

- Delete the word *trace* from the DBMS connection property in your application script.

  For example, here is the DBMS connection property in a PowerBuilder script that enables the Database Trace. (This syntax assumes you are using the default Transaction object SQLCA, but you can also define your own Transaction object.)

  ```
  SQLCA.DBMS   = "trace SYC"
  ```

  Here is how the same DBMS connection property should look after you edit it to stop tracing:

  ```
  SQLCA.DBMS   = "SYC"
  ```

Reading the DBMS value from an external text file

As an alternative to editing the DBMS property in your PowerBuilder application script, you can use the PowerScript ProfileString function to read the DBMS value from a specified section of an external text file, such as an application-specific initialization file.

This assumes that the DBMS value read from your initialization file *does not include* the word *trace*, as shown in the preceding example in Editing the DBMS property.

Using the Database Trace log

PowerBuilder writes the output of the Database Trace tool to a file named *DBTRACE.LOG* (by default) or to a nondefault log file that you specify. To use the trace log, you can do the following anytime:

- View the Database Trace log with any text editor
- Annotate the Database Trace log with your own comments
- Delete the Database Trace log or clear its contents when it becomes too large

Viewing the Database Trace log

You can display the contents of the log file anytime during a PowerBuilder session.

❖ **To view the contents of the log file:**

- Open the log file in one of the following ways:
  - Use the File Editor in PowerBuilder. (For instructions, see the *Users Guide*.)
  - Use any text editor outside PowerBuilder.

Leaving the log file open
If you leave the log file open as you work in PowerBuilder, the Database Trace tool *does not update* the log.

Annotating the Database Trace log

When you use the Database Trace log as a troubleshooting tool, it might be helpful to add your own comments or notes to the file. For example, you can specify the date and time of a particular connection, the versions of database server and client software you used, or any other useful information.

❖ **To annotate the log file:**

1. Open the *DBTRACE.LOG* file in one of the following ways:
   - Use the File Editor in PowerBuilder. (For instructions, see the *Users Guide*.)
   - Use any text editor outside PowerBuilder.

2 Edit the log file with your comments.

3 Save your changes to the log file.

Deleting or clearing the Database Trace log

Each time you connect to a database with tracing enabled, PowerBuilder appends the trace output of your connection to the existing log. As a result, the log file can become very large over time, especially if you frequently enable tracing when connected to a database.

❖ **To keep the size of the log file manageable:**

- Do either of the following periodically:
  - Open the log file, clear its contents, and save the empty file.

    Provided that you use the default *DBTRACE.LOG* or the same nondefault file the next time you connect to a database with tracing enabled, PowerBuilder will write to this empty file.
  - Delete the log file.

    PowerBuilder will automatically create a new log file the next time you connect to a database with tracing enabled.

Sample Database Trace output

This section gives an example of Database Trace output that you might see in the log file and briefly explains each portion of the output.

The example traces a connection with Sum Timing enabled. The output was generated while running a PowerBuilder application that displays information about authors in a publications database. The SELECT statement shown retrieves information from the Author table.

The precision (for example, microseconds) used when Database Trace records internal commands depends on your operating system's timing mechanism. Therefore, the timing precision in your Database Trace log might vary from this example.

Connect to database

```
CONNECT TO TRACE SYC Adaptive Server Enterprise:
DATABASE=pubs2
LOGID=bob
SERVER=HOST12
DPPARM=Release='12.5.2',StaticBind=0
```

Prepare SELECT statement

```
PREPARE:
SELECT authors.au_id, authors.au_lname, authors.state
FROM authors
WHERE ( authors.state not in ( 'CA' ) )
ORDER BY authors.au_lname ASC (3.386 MS / 20.349 MS)
```

Get column descriptions

```
DESCRIBE: (0.021 MS / 20.370 MS)
name=au_id,len=12,type=CHAR,pbt=1,dbt=1,ct=0,prec=0,
  scale=0
name=au_lname,len=41,type=CHAR,pbt=1,dbt=1,ct=0,
  prec=0,scale=0
name=state,len=3,type=CHAR,pbt=1,dbt=1,ct=0,prec=0,
  scale=0
```

Bind memory buffers to columns

```
BIND SELECT OUTPUT BUFFER (DataWindow):
  (0.007 MS / 20.377 MS)
name=au_id,len=12,type=CHAR,pbt=1,dbt=1,ct=0,prec=0,
  scale=0
name=au_lname,len=41,type=CHAR,pbt=1,dbt=1,ct=0,
  prec=0,scale=0
name=state,len=3,type=CHAR,pbt=1,dbt=1,ct=0,prec=0,
  scale=0
```

Execute SELECT statement

```
EXECUTE: (0.001 MS / 20.378 MS)
```

Fetch rows from result set

```
FETCH NEXT: (0.028 MS / 20.406 MS)
 au_id=648-92-1872 au_lname=Blotchet-Hall state=OR
FETCH NEXT: (0.012 MS / 20.418 MS)
 au_id=722-51-5454 au_lname=DeFrance state=IN
...
FETCH NEXT: (0.010 MS / 20.478 MS)
au_id=341-22-1782 au_lname=Smith state=KS
FETCH NEXT: (0.025 MS / 20.503 MS)
*** DBI_FETCHEND *** (rc 100)
```

Update and commit database changes

```
PREPARE:
UPDATE authors SET state = 'NM'
WHERE au_id = '648-92-1872' AND au_lname = 'Blotchet-
Halls' AND state = 'OR'  (3.284 MS / 23.787 MS)
EXECUTE: (0.001 MS / 23.788 MS)
GET AFFECTED ROWS: (0.001 MS / 23.789 MS)
^ 1 Rows Affected
COMMIT: (1.259 MS / 25.048 MS)
```

Disconnect from database

```
DISCONNECT: (0.764 MS / 25.812 MS)
```

Shut down database interface

```
SHUTDOWN DATABASE INTERFACE: (0.001 MS / 25.813 MS)
```

Using the SQL statement trace utility

SQL statement tracing

A separate database trace utility lets you add date and time entries to a log file for each SQL statement issued to the database, along with the syntax of the SQL statement. By default, this utility saves all log entries to a file named *PBTRSQL.log* in the initialization path directory. You can set the initialization path in the on the General tab of the System Options dialog box.

You can also change the log file location and log file name in the registry or in the DbTrace section of the *PB.INI* file in the same way you change the trace output file name for the main database trace utility (see INI file settings for DBTrace on page 206):

```
[DbTrace]
SqlTraceFile=c:\myApplication\tracesql.log
```

The registry string for the log file name is SqlTraceFile. It is located under the *HKEY_CURRENT_USER\Software\Sybase\PowerBuilder\17.0\DBTrace* key. If the DbTrace section in the PB.INI file has at least one entry, the registry value is ignored. The default file name is used only if both the registry value and the *PB.INI* value are not set.

You start the SQL statement trace utility in PowerScript code by invoking the driver for the DBMS that you want to use with a TRS modifier. You set the driver in the DBMS property of a connection object. For example, for the default SQLCA connection object, if you wanted to use ODBC with SQL tracing, you would code the following:

```
SQLCA.DBMS="TRS ODBC"
```

You can start and stop the SQL statement trace utility in PowerScript in the same way you start and stop the main database utility: you can start trace logging by setting the DBParm parameter to “PBTrace=1” and you can stop trace logging by setting the parameter to “PBTrace=0”.

For more information, see Starting a trace in PowerScript with the PBTrace parameter on page 211.

Server-side timestamps

Server-side timestamps can be used instead of client-side timestamps if the connecting PowerBuilder database driver supports the DBI_GET_SERVER_TIME command type. Currently, server-side timestamps are available for the ASE, SYC, and ODBC drivers.

PBTRS170.DLL obtains the date and time from the server only once during the database connection processing. Each time a new timestamp needs to be generated, it determines the number of milliseconds that have transpired since the connection was established and computes the new server-side date and time by adding the elapsed interval to the initial connection timestamp obtained from the server.

Log file headers

Output to the log file is always appended. For ease of reading, the *PBTRS170.DLL* produces a banner inside the log file each time a new database connection is established. The banner lists the date and time of the database connection using the system clock on the client workstation. The DBParms for the database connection are listed immediately under the banner. If a server timestamp is used for subsequent entries in the log file, the statement "Using timestamp from DBMS server" is entered immediately under the DBParm listings.

When you are running an application with a database trace utility, one of the DBParm values should include the DisableBind parameter. You should set DisableBind to 1. Otherwise the syntax that is logged in the trace output file will contain parameter markers instead of human-readable values.

The following output shows a banner from a trace file that uses a client-side timestamp in the banner itself, and server-side timestamps elsewhere:

```
/*---------------------------------------------------*/
/*                 1/10/2007  16:08                  */
/*---------------------------------------------------*/
(60ec068): CONNECT TO TRS ODBC:  DBPARM=ConnectString='DSN= PB
Demo DB V2017;UID=dba;PWD=sql'  SERVER=DemoDatabase170
(60ec068): Using timestamp from DBMS server. (1/10/2007
16:08:28.079)
(60ec068): PREPARE:  (1/10/2007  16:08:44.513)   SELECT
DISTINCT  "pbcattbl"."pbt_tnam" , "pbcattbl"."pbt_cmnt"
FROM "pbcattbl"   ORDER BY "pbcattbl"."pbt_tnam"    ASC
```

Using the ODBC Driver Manager Trace tool

This section describes how to use the ODBC Driver Manager Trace tool.

About ODBC Driver Manager Trace

You can use the ODBC Driver Manager Trace tool to trace a connection to any ODBC data source that you access in PowerBuilder through the ODBC interface.

Unlike the Database Trace tool, the ODBC Driver Manager Trace tool *cannot* trace connections through one of the native database interfaces.

What this tool does

ODBC Driver Manager Trace records information about ODBC API calls (such as SQLDriverConnect, SQLGetInfo, and SQLFetch) made by PowerBuilder while connected to an ODBC data source. It writes this information to a default log file named *SQL.LOG* or to a log file that you specify.

What both tools do

The information from ODBC Driver Manager Trace, like Database Trace, can help you:

- Understand what PowerBuilder is doing *internally* while connected to an ODBC data source
- Identify and resolve problems with your ODBC connection
- Provide useful information to Technical Support if you call them for help with your database connection

When to use this tool

Use ODBC Driver Manager Trace *instead* of the Database Trace tool if you want more detailed information about the ODBC API calls made by PowerBuilder.

Performance considerations
Turning on ODBC Driver Manager Trace can slow your performance while working in PowerBuilder. Therefore, use ODBC Driver Manager Trace for debugging purposes only and keep it turned off when you are not debugging.

SQL.LOG file

PowerBuilder writes ODBC Driver Manager Trace output to a default log file named *SQL.LOG* or to a log file that you specify. The default location of *SQL.LOG* is in your root directory.

Starting ODBC Driver Manager Trace

By default, ODBC Driver Manager Trace is turned off in PowerBuilder. You can start it in order to trace your ODBC connection in two ways:

- Edit your database profile in the PowerBuilder development environment
- Edit a script in a PowerBuilder application

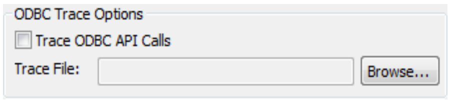
Starting ODBC Driver Manager Trace in the development environment

To start ODBC Driver Manager Trace in the PowerBuilder development environment, edit the database profile for the connection you want to trace, as described in the following procedure.

❖ To start ODBC Driver Manager Trace by editing the database profile:

1. Open the Database Profile Setup-ODBC dialog box for the ODBC connection you want to trace.
2. On the Options tab, select the Trace ODBC API Calls check box.
3. (Optional) To specify a log file where you want PowerBuilder to write the output of ODBC Driver Manager Trace, type the path name in the Trace File box
   or
   (Optional) Click Browse to display the pathname of an existing log file in the Trace File box.

   By default, if the Trace ODBC API Calls check box is selected and no trace file is specified, PowerBuilder sends ODBC Driver Manager Trace output to the default *SQL.LOG* file.

4. Click OK or Apply
   or
   Right-click on the connected database and select Re-connect from the drop-down menu in the Database Profiles dialog box.

   The Database Profiles dialog box displays with the name of the edited profile highlighted.

   PowerBuilder saves your settings in the database profile entry in the registry in the *HKEY_CURRENT_USER\Software\Sybase\17.0\DatabaseProfiles* key.

For example, here is the relevant portion of a database profile entry for an ODBC data source named Employee. The settings that start ODBC Driver Manager Trace (corresponding to the ConnectOption DBParm parameter) are emphasized.

```
DBMS     "ODBC"
...
DbParm   "ConnectString='DSN=Emloyee;UID=dba;
PWD=00c61737',ConnectOption='SQL_OPT_TRACE,SQL_OPT_
TRACE_ON;SQL_OPT_TRACEFILE,C:\Temp\odbctrce.log'”
```

5 Click Connect in the Database Profiles dialog box to connect to the database
or
Right-click on the connected database and select Re-connect from the drop-down menu in the Database Profiles dialog box.

PowerBuilder connects to the database, starts tracing the ODBC connection, and writes output to the log file you specified.

Starting ODBC Driver Manager Trace in a PowerBuilder application

To start ODBC Driver Manager Trace in a PowerBuilder application, you must specify certain values for the ConnectOption DBParm parameter in the appropriate script. For example, you might include them in the script that opens the application.

You can specify the required ConnectOption values in a PowerBuilder script by:

- (*Recommended*) Copying the PowerScript ConnectOption DBParm syntax from the Preview tab in the Database Profile Setup dialog box into your script
- Coding PowerScript to set a value for the DBParm property of the Transaction object
- Reading the DBParm values from an external text file

For more about using Transaction objects to communicate with a database in a PowerBuilder application, see *Application Techniques*.

About the ConnectOption DBParm parameter

ConnectOption includes several parameters, two of which control the operation of ODBC Driver Manager Trace for any ODBC-compatible driver you are using in PowerBuilder.

Table 15-2: ConnectOption parameters for ODBC Driver Manager Trace

Parameter	Description
SQL_OPT_TRACE	**Purpose** Starts or stops ODBC Driver Manager Trace in PowerBuilder. **Values** The values you can specify are: • **SQL_OPT_TRACE_OFF** (Default) Stops ODBC Driver Manager Trace • **SQL_OPT_TRACE_ON** Starts ODBC Driver Manager Trace
SQL_OPT_TRACEFILE	**Purpose** Specifies the name of the trace file where you want to send the output of ODBC Driver Manager Trace. PowerBuilder appends the output to the trace file you specify until you stop the trace. To display the trace file, you can use the File Editor (in PowerBuilder) or any text editor (outside PowerBuilder). **Values** You can specify any filename for the trace file, *following the naming conventions of your operating system*. By default, if tracing is on and you have not specified a trace file, PowerBuilder sends ODBC Driver Manager Trace output to a file named *SQL.LOG*. For information about the location of *SQL.LOG* on different platforms, see About ODBC Driver Manager Trace on page 219.

Copying ConnectOption syntax from the Preview tab

The easiest way to start ODBC Driver Manager Trace in a PowerBuilder application script is to copy the PowerScript ConnectString DBParm syntax from the Preview tab in the Database Profile Setup - ODBC dialog box into your script, modifying the default Transaction object name (SQLCA) if necessary.

As you complete the Database Profile Setup dialog box in the development environment, PowerBuilder generates the correct connection syntax on the Preview tab. Therefore, copying the syntax directly from the Preview tab into your script ensures that it is accurate.

❖ **To copy ConnectOption syntax from the Preview tab into your script:**

1. On the Options tab in the Database Profile Setup - ODBC dialog box for your connection, select the Trace ODBC API Calls check box and (optionally) specify a log file in the Trace File box to start ODBC Driver Manager Trace.
2. Click Apply to save your changes to the Options tab without closing the dialog box.
3. Click the Preview tab.

   The correct PowerScript syntax for ODBC Driver Manager Trace and other selected options displays in the Database Connection Syntax box.

   The following example shows the PowerScript syntax that starts ODBC Driver Manager Trace and sends output to the file *C:\TEMP\ODBCTRCE.LOG*.

```
// Profile Employee
SQLCA.DBMS = "ODBC"
SQLCA.AutoCommit = False
SQLCA.DBParm = "Connectstring='DSN=Employee',
   ConnectOption='SQL_OPT_TRACE,SQL_OPT_TRACE_ON;
   SQL_OPT_TRACEFILE,c:\temp\odbctrce.log’"
```

4. Select the SQLCA.DBParm line and any other syntax you want to copy to your script and click Copy.

   PowerBuilder copies the selected text to the clipboard.

5. Paste the selected text from the Preview tab into your script, modifying the default Transaction object name (SQLCA) if necessary.

Coding PowerScript to set a value for the DBParm property

Another way to start ODBC Driver Manager Trace in a PowerBuilder application script is to include the ConnectOption parameters that control tracing as values for the DBParm property of the Transaction object.

❖ **To start ODBC Driver Manager Trace by setting the DBParm property:**

- In your application script, set the SQL_OPT_TRACE and (optionally) SQL_OPT_TRACEFILE ConnectOption parameters to start the trace and to specify a nondefault trace file, respectively.

  For example, the following statement starts ODBC Driver Manager Trace in your application and sends output to a file named *MYTRACE.LOG*. Insert a comma to separate the ConnectString and ConnectOption values.

  This example assumes you are using the default Transaction object SQLCA, but you can also define your own Transaction object.

```
SQLCA.DBParm="ConnectString='DSN=Test;UID=PB;
   PWD=xyz',ConnectOption='SQL_OPT_TRACE,
   SQL_OPT_TRACE_ON;SQL_OPT_TRACEFILE,C:\TRC.LOG'"
```

Reading the DBParm value from an external text file

As an alternative to setting the DBParm property in your PowerBuilder application script, you can use the PowerScript ProfileString function to read DBParm values from a specified section of an external text file, such as an application-specific initialization file.

This assumes that the DBParm value read from your initialization file includes the ConnectOption parameter to start ODBC Driver Manager Trace, as shown in the preceding example.

❖ **To start ODBC Driver Manager Trace by reading DBParm values from an external text file:**

- Use the following PowerScript syntax to specify the ProfileString function with the DBParm property:

  SQLCA.dbParm = ProfileString(*file*, *section*, *variable*, *default_value*)

  For example, the following statement in a PowerBuilder script reads the DBParm values from the [Database] section of the *APP.INI* file:

  ```
  SQLCA.dbParm =
     ProfileString("APP.INI","Database","DBParm","")
  ```

Stopping ODBC Driver Manager Trace

Once you start tracing an ODBC connection with ODBC Driver Manager Trace, PowerBuilder continues sending trace output to the log file until you stop tracing. After you stop tracing as described in the following sections, you must reconnect to have the changes take effect.

Stopping ODBC Driver Manager Trace in the development environment

❖ **To stop ODBC Driver Manager Trace by editing a database profile:**

1. Open the Database Profile Setup - ODBC dialog box for the connection you are tracing.

   For instructions, see Starting ODBC Driver Manager Trace on page 220.

2. On the Options tab, clear the Trace ODBC API Calls check box.

If you supplied the pathname of a log file in the Trace File box, you can leave it specified in case you want to restart tracing later.

3 Click OK in the Database Profile Setup - ODBC dialog box.

The Database Profiles dialog box displays, with the name of the edited profile highlighted.

4 Click Connect in the Database Profiles dialog box or right-click on the connected database and select Re-connect from the drop-down menu in the Database Profiles dialog box.

PowerBuilder connects to the database and stops tracing the connection.

Stopping ODBC Driver Manager Trace in a PowerBuilder application

To stop ODBC Driver Manager Trace in a PowerBuilder application script, you must change the SQL_OPT_TRACE ConnectOption parameter to SQL_OPT_TRACE_OFF. You can do this by:

- Editing the value of the DBParm property of the Transaction object
- Reading the DBParm values from an external text file

Editing the DBParm property

One way to change the ConnectOption value in a PowerBuilder script is to edit the DBParm property of the Transaction object.

❖ **To stop ODBC Driver Manager Trace by editing the DBParm property:**

- In your application script, edit the DBParm property of the Transaction object to change the value of the SQL_OPT_TRACE ConnectOption parameter to SQL_OPT_TRACE_OFF.

  For example, the following statement starts ODBC Driver Manager Trace in your application and sends the output to a file named *MYTRACE.LOG*. (This example assumes you are using the default Transaction object SQLCA, but you can also define your own Transaction object.)

```
SQLCA.DBParm="ConnectString='DSN=Test;UID=PB;
   PWD=xyz',ConnectOption='SQL_OPT_TRACE,
   SQL_OPT_TRACE_ON;SQL_OPT_TRACEFILE,C:\TRC.LOG'"
```

Here is how the same statement should look after you edit it to stop ODBC Driver Manager Trace. (You can leave the name of the trace file specified in case you want to restart tracing later.)

```
SQLCA.DBParm="ConnectString='DSN=Test;UID=PB;
   PWD=xyz',ConnectOption='SQL_OPT_TRACE,
   SQL_OPT_TRACE_OFF;SQL_OPT_TRACEFILE,C:\TRC.LOG'"
```

Reading DBParm values

As an alternative to editing the DBParm property in your PowerBuilder application script, you can use the PowerScript ProfileString function to read DBParm values from a specified section of an external text file, such as an application-specific initialization file.

This assumes that the DBParm value read from your initialization file sets the value of SQL_OPT_TRACE to SQL_OPT_TRACE_OFF, as shown in the preceding example.

Viewing the ODBC Driver Manager Trace log

You can display the contents of the ODBC Driver Manager Trace log file anytime during a PowerBuilder session.

Location of SQL.LOG
For information about where to find the default SQL.LOG file, see About ODBC Driver Manager Trace on page 219.

❖ **To view the contents of the log file:**

- Open SQL.LOG or the log file you specified in one of the following ways:
  - Use the File Editor in PowerBuilder. (For instructions, see the *Users Guide*.)
  - Use any text editor outside PowerBuilder.

Leaving the log file open
If you leave the log file open as you work in PowerBuilder, ODBC Driver Manager Trace *does not update it*.

Sample ODBC Driver Manager Trace output

This section shows a partial example of output from ODBC Driver Manager Trace to give you an idea of the information it provides. The example is part of the trace on an ODBC connection to the Demo Database.

For more about a particular ODBC API call, see your ODBC documentation.

```
PB125 179:192   EXIT  SQLSetConnectOption  with return
code 0 (SQL_SUCCESS)
      HDBC 0x036e1300
      UWORD    104 <SQL_OPT_TRACE>
      UDWORD 1

PB125 179:192   EXIT  SQLGetInfoW  with return code 0
(SQL_SUCCESS)
      HDBC 0x036e1300
      UWORD 25 <SQL_DATA_SOURCE_READ_ONLY>
      PTR 0x036e3c88 [       2] "N"
      SWORD 512
      SWORD * 0x0012cc32 (2)
```

Using the JDBC Driver Manager Trace tool

This section describes how to use the JDBC Driver Manager Trace tool.

About JDBC Driver Manager Trace

You can use the JDBC Driver Manager Trace tool to trace a connection to any database that you access in PowerBuilder through the JDBC interface.

Unlike the Database Trace tool, the JDBC Driver Manager Trace tool *cannot* trace connections through one of the native database interfaces.

What this tool does

JDBC Driver Manager Trace logs errors and informational messages originating from the Driver object currently loaded (such as SAP's jConnect JDBC driver) when PowerBuilder connects to a database through the JDBC interface. It writes this information to a default log file named *JDBC.LOG* or to a log file that you specify. The amount of trace output varies depending on the JDBC driver being used.

What both tools do

The information from JDBC Driver Manager Trace, like Database Trace, can help you:

- Understand what PowerBuilder is doing *internally* while connected to a database through the JDBC interface
- Identify and resolve problems with your JDBC connection
- Provide useful information to Technical Support if you call them for help with your database connection

When to use this tool

Use JDBC Driver Manager Trace *instead* of the Database Trace tool if you want more detailed information about the JDBC driver.

Performance considerations
Turning on JDBC Driver Manager Trace can slow your performance while working in PowerBuilder. Therefore, use JDBC Driver Manager Trace for debugging purposes only and keep it turned off when you are not debugging.

JDBC.LOG file

PowerBuilder writes JDBC Driver Manager Trace output to a default log file named *JDBC.LOG* or to a log file that you specify. The default location of *JDBC.LOG* is a temp directory.

Starting JDBC Driver Manager Trace

By default, JDBC Driver Manager Trace is turned off in PowerBuilder. You can start it in order to trace your JDBC connection in two ways:

- Edit your database profile in the PowerBuilder development environment
- Edit a script in a PowerBuilder application

Starting JDBC Driver Manager Trace in the development environment

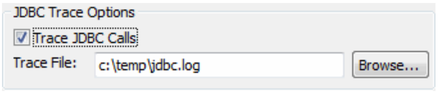
To start JDBC Driver Manager Trace in the PowerBuilder development environment, edit the database profile for the connection you want to trace, as described in the following procedure.

❖ **To start JDBC Driver Manager Trace by editing the database profile:**

1. Open the Database Profile Setup - JDBC dialog box for the JDB connection you want to trace.
2. On the Options tab, select the Trace JDBC Calls check box.

3 (Optional) To specify a log file where you want PowerBuilder to write the output of JDBC Driver Manager Trace, type the path name in the Trace File box, or click Browse to display the path name of an existing log file in the Trace File box.

By default, if the Trace JDBC Calls check box is selected and no alternative trace file is specified, PowerBuilder sends JDBC Driver Manager Trace output to the default *JDBC.LOG* file.


JDBC Trace Options

☑ Trace JDBC Calls

Trace File: c:\temp\jdbc.log

4 Click OK or Apply.

The Database Profiles dialog box displays with the name of the edited profile highlighted. PowerBuilder saves your settings in the database profile entry in the registry.

For example, here are the DBMS and DBParm string values of a database profile entry for a database named Employee. The settings that start JDBC Driver Manager Trace (corresponding to the TraceFile DBParm parameter) are emphasized.

```
DBMS     "TRACE JDBC"
DbParm   "Driver='com.sybase.jdbc3.jdbc.SybDriver',
          URL='jdbc:sybase:Tds:199.93.178.151:
          5007/tsdata',TraceFile='c:\temp\jdbc.log'"
```

5 Click Connect in the Database Profiles dialog box to connect to the database
or
Right-click on the connected database and select Re-connect from the drop-down menu in the Database Profiles dialog box.

PowerBuilder connects to the database, starts tracing the JDBC connection, and writes output to the log file you specified.

Starting JDBC Driver Manager Trace in a PowerBuilder application

To start JDBC Driver Manager Trace in a PowerBuilder application, you must specify the TraceFile DBParm parameter in the appropriate script. For example, you might include it in the script that opens the application.

You can specify the TraceFile parameter in a PowerBuilder script by:

- (*Recommended*) Copying the PowerScript TraceFile DBParm syntax from the Preview tab in the Database Profile Setup dialog box into your script
- Coding PowerScript to set a value for the DBParm property of the Transaction object
- Reading the DBParm values from an external text file

For more about using Transaction objects to communicate with a database in a PowerBuilder application, see *Application Techniques*.

About the TraceFile DBParm parameter

TraceFile controls the operation of JDBC Driver Manager Trace for any JDBC-compatible driver you are using in PowerBuilder.

Copying TraceFile syntax from the Preview tab

The easiest way to start JDBC Driver Manager Trace in a PowerBuilder application script is to copy the PowerScript TraceFile DBParm syntax from the Preview tab in the Database Profile Setup - JDBC dialog box into your script, modifying the default Transaction object name (SQLCA) if necessary.

As you complete the Database Profile Setup dialog box in the development environment, PowerBuilder generates the correct connection syntax on the Preview tab. Therefore, copying the syntax directly from the Preview tab into your script ensures that it is accurate.

❖ To copy TraceFile syntax from the Preview tab into your script:

1. On the Options tab in the Database Profile Setup - JDBC dialog box for your connection, select the Trace JDBC Calls check box and (optionally) specify a log file in the Trace File box to start JDBC Driver Manager Trace.

   For instructions, see Stopping JDBC Driver Manager Trace in the development environment on page 232.

2. Click Apply to save your changes to the Options tab without closing the dialog box.

3. Click the Preview tab.

   The correct PowerScript syntax for JDBC Driver Manager Trace and other selected options displays in the Database Connection Syntax box.

The following example shows the PowerScript syntax that starts JDBC Driver Manager Trace and sends output to the file *C:\TEMP\JDBC.LOG*.

```
// Profile Employee
SQLCA.DBMS = "TRACE JDBC"
SQLCA.DBParm =
"Driver='com.sybase.jdbc3.jdbc.SybDriver',
URL='jdbc:sybase:Tds:199.93.178.151:5007/tsdata',
TraceFile='c:\temp\jdbc.log'"
```

4. Select the DBParm line and any other syntax you want to copy to your script and click Copy.

   PowerBuilder copies the selected text to the clipboard.

5. Paste the selected text from the Preview tab into your script, modifying the default Transaction object name (SQLCA) if necessary.

Coding PowerScript to set a value for the DBParm property

Another way to start JDBC Driver Manager Trace in a PowerBuilder application script is to include the TraceFile parameter as a value for the DBParm property of the Transaction object.

❖ **To start JDBC Driver Manager Trace by setting the DBParm property:**

- In your application script, include the TraceFile parameter to start the trace and specify a nondefault trace file.

  For example, this statement starts JDBC Driver Manager Trace in your application and sends output to a file named *MYTRACE.LOG*. (This example assumes you are using the default Transaction object SQLCA, but you can also define your own Transaction object.)

```
SQLCA.DBParm =
"Driver='com.sybase.jdbc3.jdbc.SybDriver',
URL='jdbc:sybase:Tds:199.93.178.151:5007/tsdata',
TraceFile='c:\MYTRACE.LOG'"
```

Reading the DBParm value from an external text file

As an alternative to setting the DBParm property in your PowerBuilder application script, you can use the PowerScript ProfileString function to read DBParm values from a specified section of an external text file, such as an application-specific initialization file.

This assumes that the DBParm value read from your initialization file includes the TraceFile parameter to start JDBC Driver Manager Trace, as shown in the preceding example.

❖ **To start JDBC Driver Manager Trace by reading DBParm values from an external text file:**

- Use the following PowerScript syntax to specify the ProfileString function with the DBParm property:

  SQLCA.dbParm = ProfileString(*file*, *section*, *variable*, *default_value*)

  For example, the following statement in a PowerBuilder script reads the DBParm values from the [Database] section of the *APP.INI* file:

```
SQLCA.dbParm =
   ProfileString("APP.INI","Database","DBParm","")
```

Stopping JDBC Driver Manager Trace

Once you start tracing a JDBC connection with JDBC Driver Manager Trace, PowerBuilder continues sending trace output to the log file until you stop tracing.

Stopping JDBC Driver Manager Trace in the development environment

❖ **To stop JDBC Driver Manager Trace by editing a database profile:**

1. Open the Database Profile Setup - JDBC dialog box for the connection you are tracing.

   For instructions, see Starting JDBC Driver Manager Trace on page 228.

2. On the Options tab, clear the Trace JDBC Calls check box.

   If you supplied the path name of a log file in the Trace File box, you can leave it specified in case you want to restart tracing later.

3. Click OK in the Database Profile Setup - JDBC dialog box.

   The Database Profiles dialog box displays, with the name of the edited profile highlighted.

4. Click Connect in the Database Profiles dialog box or right click on the connected database and select Re-connect from the drop-down menu in the Database Profiles dialog box.

   PowerBuilder connects to the database and stops tracing the connection.

Stopping JDBC Driver Manager Trace in a PowerBuilder application

To stop JDBC Driver Manager Trace in a PowerBuilder application script, you must delete the TraceFile parameter. You can do this by:

- Editing the value of the DBParm property of the Transaction object
- Reading the DBParm values from an external text file

Editing the DBParm property

One way to change the TraceFile parameter in a PowerBuilder script is to edit the DBParm property of the Transaction object.

❖ **To stop JDBC Driver Manager Trace by editing the DBParm property:**

- In your application script, edit the DBParm property of the Transaction object to delete the TraceFile parameter.

  For example, the following statement starts JDBC Driver Manager Trace in your application and sends the output to a file named *MYTRACE.LOG*. (This example assumes you are using the default Transaction object SQLCA, but you can also define your own Transaction object.)

  ```
  SQLCA.DBParm =
  "Driver='com.sybase.jdbc3.jdbc.SybDriver',
  URL='jdbc:sybase:Tds:199.93.178.151:5007/tsdata',
  TraceFile='c:\MYTRACE.LOG'"
  ```

  Here is how the same statement should look after you edit it to stop JDBC Driver Manager Trace.

  ```
  SQLCA.DBParm =
  "Driver='com.sybase.jdbc3.jdbc.SybDriver',
  URL='jdbc:sybase:Tds:199.93.178.151:5007/tsdata'"
  ```

Reading DBParm values

As an alternative to editing the DBParm property in your PowerBuilder application script, you can use the PowerScript ProfileString function to read DBParm values from a specified section of an external text file, such as an application-specific initialization file, or you can use RegistryGet to obtain values from a registry key.

This assumes that the DBParm is no longer read from your initialization file or registry key, as shown in the preceding example. You must disconnect and reconnect for this to take effect.

Viewing the JDBC Driver Manager Trace log

You can display the contents of the JDBC Driver Manager Trace log file anytime during a PowerBuilder session.

Location of JDBC.LOG
For information about where to find the default *JDBC.LOG* file, see About JDBC Driver Manager Trace on page 227.

❖ **To view the contents of the log file:**

- Open *JDBC.LOG* or the log file you specified in one of the following ways:
    - Use the File Editor in PowerBuilder. (For instructions, see the *Users Guide*.)
    - Use any text editor outside PowerBuilder.

Leaving the log file open
If you leave the log file open as you work in PowerBuilder, JDBC Driver Manager Trace *does not update the log*.

PART 6

Using Embedded SQL

This part describes how to use embedded SQL when accessing a database with that interface in a PowerBuilder application.

CHAPTER 16 Using Embedded SQL with ODBC

About this chapter

When you create scripts for a PowerBuilder application, you can use embedded SQL statements in the script to perform operations on the database. The features supported when you use embedded SQL depend on the DBMS to which your application connects.

Overview

When you use the ODBC interface to connect to a backend database, you can use embedded SQL in your scripts.

You can embed the following types of SQL statements in scripts and user-defined functions if the ODBC driver you are using and the backend DBMS you are accessing supports this functionality. (Not all backend databases support cursor statements and database stored procedures.)

- Transaction management statements
- Non-cursor statements
- Cursor statements
- Database stored procedures

ODBC API

The ODBC interface uses the ODBC application programming interface (API) to interact with the backend database.

When you use embedded SQL, PowerBuilder makes the required calls to the backend database. Therefore, you do not need to know anything about the ODBC API to use embedded SQL with PowerBuilder.

See also

Chapter 2, Using the ODBC Interface
ODBC SQL Support
ODBC Transaction management statements
ODBC Non-cursor statements
ODBC Cursor statements
ODBC Database stored procedures

ODBC SQL Support

PowerBuilder embedded SQL supports the name qualification conventions and functions used in the databases accessible through the ODBC interface.

See also

ODBC Name qualification
ODBC SQL functions

ODBC Name qualification

PowerBuilder does not inspect all SQL statement syntax, so you can qualify database catalog entities as necessary.

For example, the following qualifications are acceptable for an ODBC interface to a SQL Anywhere database:

- emp_name
- employee.emp_name

ODBC SQL functions

In SQL statements, you can use any function that your backend DBMS supports (such as aggregate or mathematical functions). For example, if your DBMS supports the function Sum, you can use the function Sum in a SELECT statement:

```
SELECT Sum(salary)
    INTO :salary_sum_var
    FROM employee;
```

Calling ODBC functions

While PowerBuilder provides access to a large percentage of the features within ODBC, in some cases you may decide that you need to call one or more ODBC functions directly for a particular application. PowerBuilder provides access to most Windows DLLs by using external function declarations.

The ODBC calls qualify for this type of access. Most ODBC calls require a pointer to a connection handle (of the variable type HDBC) to a structure as their first parameter. If you want to call ODBC without reconnecting to the database to get a connection handle, use the PowerScript DBHandle function.

DBHandle

DBHandle takes a transaction object as a parameter and returns a long variable, which is the handle to the database for the transaction. This handle is actually the connection handle PowerBuilder uses internally to communicate with the database. You can use this returned long value in the ODBC DLLs and pass it as one of the parameters in your function.

After you obtain the connection handle, you can use the ODBC SQLGetInfo call to obtain the environment handle of the variable type HENV.

Example

This example illustrates how to use DBHandle. As with other examples, assume a successful connection has occurred using the default transaction object (SQLCA).

```
// Define a variable to hold the DB connection
// handle
long  ODBCConnectionHandle

// Go get the handle.
ODBCConnectionHandle = SQLCA.DBHandle( )

// Now that you have the ODBC connection pointer,
// call the DLL function.
MyDLLFunction(ODBCConnectionHandle, parm1, parm2)
```

In your DLL, cast the incoming connection handle of the type HDBC:

```
MyDLLFunction(long lODBCConnectionHandle,
    parm1_type parm1,
    parm2_type Parm2, ...)
{
HDBC * pDatabase;
pDatabase = (HDBC *)  lODBCConnectionHandle;
// ODBC functions can be called using pDatabase.
}
```

See also

ODBC Using escape clauses

ODBC Using escape clauses

ODBC defines extensions that are common to most backend DBMSs. To cover vendor-specific extensions, the syntax defined by ODBC uses the escape clause provided by the X/Open and SQL Access Group (SAG) SQL draft specifications.

For example, some of the extensions defined in ODBC are:

- Date, time, and timestamp data
- Scalar functions (such as data type, numeric, and string conversion functions)
- Outer joins
- Procedures

Maximum portability For maximum portability, you should use escape sequences in your applications.

Syntax For example, PowerBuilder uses the date, time, and timestamp escape clauses as the default formats for data manipulation. The syntax for each of these escape clauses is:

```
{ d yyyy-mm-dd }
```

```
{ t hh:mm:ss }
```

```
{ ts yyyy-mm-dd hh:mm:ss:[fff[fff]] }
```

Example Each of the following statements updates employee Henry Jones's start time in the Employee table. The first statement uses the escape clause, and the second statement uses native syntax for a time column:

```
UPDATE Employee
    SET start_time = {t 08:30:00}
    WHERE emp_name = "Henry Jones"
UPDATE Employee
    SET start_time = (08:30:00)
    WHERE emp_name = "Henry Jones"
```

ODBC Transaction management statements

If the database you are connecting to supports transaction management, you can use the following transaction management statements with one or more transaction objects to manage connections and transactions for a database:

- CONNECT
- DISCONNECT
- COMMIT
- ROLLBACK

See also ODBC Using CONNECT, DISCONNECT, COMMIT, and ROLLBACK

ODBC Using CONNECT, DISCONNECT, COMMIT, and ROLLBACK

The following table lists each transaction management statement and describes how it works when you use the ODBC interface to connect to a database:

Statement	Description
CONNECT	Establishes the database connection. After you assign values to the required properties of the transaction object, you can execute a CONNECT. When you connect to the database, the DBMS name returned by the ODBC SQLGetInfo call is returned in the transaction object property SQLReturnData.
DISCONNECT	Terminates a successful connection. When a DISCONNECT is executed, PowerBuilder internally executes a COMMIT WORK statement to commit all changes and then issues a CLOSE DATABASE statement to terminate the logical unit of work.
COMMIT	Applies all changes made to the database since the beginning of the current unit of work.
ROLLBACK	Undoes all changes made to the database since the beginning of the current logical unit of work.

See also ODBC Performance and locking

ODBC Performance and locking

After a connection is established, SQL statements can cause locks to be placed on database entities. The more locks there are in place at a given moment in time, the more likely it is that the locks will hold up another transaction.

Rules

No set of rules for designing a database application is totally comprehensive. However, when you design a PowerBuilder application, you should do the following:

- **Long-running connections** Determine whether you can afford to have long-running connections. If not, your application should connect to the database only when absolutely necessary. After all the work for that connection is complete, the transaction should be disconnected.

  If long-running connections are acceptable, then COMMITs should be issued as often as possible to guarantee that all changes do in fact occur. More importantly, COMMITs should be issued to release any locks that may have been placed on database entities as a result of the statements executed using the connection.

- **SetTrans or SetTransObject function** Determine whether you want to use default DataWindow transaction processing (the SetTrans function) or control the transaction in a script (the SetTransObject function).

  If you cannot afford to have long-running connections and therefore have many short-lived transactions, use the default DataWindow transaction processing. If you want to keep connections open and issue periodic COMMITs, use the SetTransObject function and control the transaction yourself.

Switching during a connection
To switch between transaction processing and AutoCommit during a connection, change the setting of AutoCommit in the transaction object.

Isolation feature

ODBC uses the isolation feature to support assorted database lock options. In PowerBuilder, you can use the Lock property of the transaction object to set the isolation level when you connect to the database.

The following example shows how to set the Lock property to RU (Read uncommitted):

```
// Set the lock property to read uncommitted
// in the default transaction object SQLCA.
SQLCA.Lock = "RU"
```

PowerBuilder uses the ODBC API call SQ2.SetConnectOption (SetIsolationLevel) to set the isolation level. The lock value is passed to the function as a 32-bit mask.

Example 1

This script uses embedded SQL to connect to a database and attempts to insert a row in the ORDER_HEADER table and a row in the ORDER_ITEM table. The script then executes a COMMIT or ROLLBACK depending on the success of all statements in the script.

```
// Set the SQLCA connection properties.
SQLCA.DBMS = "ODBC"
SQLCA.DBParm = "connectstring = 'DSN = orders'"

// Connect to the database.
CONNECT USING SQLCA;

// Insert a row into the ORDER_HEADER table.
INSERT INTO ORDER_HEADER (ORDER_ID,CUSTOMER_ID)
    VALUES (7891, 129);

// Test return code for ORDER_HEADER insertion.
// A ROLLBACK is required only if the first row
// was inserted successfully.
if SQLCA.sqlcode = 0 then

// Since the ORDER_HEADER is inserted,
// try to insert ORDER_ITEM.
    INSERT INTO ORDER_ITEM
        (ORDER_ID, ITEM_NBR, PART_NBR, QTY)
        VALUES (7891, 1, '991PLS', 456);
// Test return code for ORDER_ITEM insertion.
    if SQLCA.sqlcode = -1 then

// Disconnect from the database.
DISCONNECT USING SQLCA;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

This example uses scripts for the Open and Close events for a window and the Clicked event for a CommandButton to illustrate how you can manage transactions for a DataWindow control. Assume a window contains a DataWindow control dw_1 and a CommandButton Cb_Update. Also assume the user enters data in dw_1 and then clicks the Cb_Update button to update the database with the data.

The window OPEN event script:

```
// Set the transaction object properties
// and connect to the database.
// Set the SQLCA connection properties.
SQLCA.DBMS = "ODBC"
SQLCA.DBParm = "connectstring = 'DSN = orders'"

// Connect to the database.
CONNECT USING SQLCA;

// Tell the DataWindow which transaction object
// to use.
dw_1.SetTransObject(sqlca)
```

The CommandButton CLICKED event script:

```
// Declare ReturnValue an integer.
integer ReturnValue
ReturnValue = dw_1.Update( )

// Test to see if updates were successful.
if ReturnValue = -1 then

// Updates were not successful. Since we used
// SetTransObject, roll back any changes made
// to the database.
   ROLLBACK USING SQLCA;
else

// Updates were successful. Since we used
// SetTransObject, commit any changes made
// to the database.
   COMMIT USING SQLCA;
end if
```

The window CLOSE event script:

```
// Disconnect from the database.
DISCONNECT USING SQLCA;
```

ODBC Non-cursor statements

The statements that do not involve cursors are:

- DELETE (ODBC DELETE, INSERT, and UPDATE)
- INSERT (ODBC DELETE, INSERT, and UPDATE)
- UPDATE (ODBC Update)
- ODBC SELECT (singleton)

ODBC DELETE, INSERT, and UPDATE

Internally, PowerBuilder processes DELETE, INSERT, and UPDATE the same way. PowerBuilder inspects these statements for variable references and replaces all variable references with a constant that conforms to the backend database's rules for that data type.

Example

Assume you enter the following statement:

```
DELETE FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var is a PowerScript variable with the data type of integer that has been defined within the scope of the script that contains the DELETE statement.

Before the DELETE statement is executed, `emp_id_var` is assigned a value (say 691) so when the DELETE statement executes, the database receives the following command:

```
DELETE FROM employee WHERE emp_id = 691;
```

When is this substitution technique used?

This variable substitution technique is used for all PowerScript variable types. When you use embedded SQL, precede all PowerScript variables with a colon (:).

See also

ODBC SELECT

ODBC SELECT

The SELECT statement contains input and output variables.

- **Input variables** are passed to the database as part of the execution, and the substitution is as described for DELETE, INSERT, and UPDATE.
- **Output variables** return values based on the result of the SELECT statement.

Example 1

Assume you enter the following statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var, emp_salary_var, and emp_name_var are PowerScript variables defined within the scope of the script containing the SELECT statement, and emp_id_var is an input variable and is processed as described in the DELETE example above.

Both emp_name_var and emp_salary_var are output variables that will be used to return values from the database. The data types of emp_name_var and emp_salary_var should be the PowerScript data types that best match the data type in the database. When the data types do not match perfectly, PowerBuilder converts them.

How big should numeric output variables be?
For numeric data, the output variable must be large enough to hold any value that may come from the database.

Assume the value for emp_id_var is 691 as in the previous example. When the SELECT statement executes, the database receives this command:

```
SELECT emp_name, emp_salary
    FROM employee WHERE emp_id = 691;
```

If no errors are returned when the statement executes, data locations are bound internally for the result fields. The data returned into these locations is converted if necessary, and the appropriate PowerScript variables are set to those values.

Example 2

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. It also assumes the data type of the emp_id column in the employee table is CHARACTER[10]. The user enters an employee ID into the single line edit field sle_Emp and clicks the button Cb_Delete.

The script for the Clicked event in the CommandButton Cb_Delete is:

```
// Make sure we have a value.
if sle_Emp.text <> "" then

// Since we have a value, let's try to delete
// it.
    DELETE FROM employee
    WHERE emp_id = :sle_Emp.text;

// Test to see if the DELETE worked.
    if SQLCA.sqlcode = 0 then

// It seems to have worked; let user know.
    MessageBox("Delete",&
        "The delete has been successfully "&
```

```
         +"processed!")
      else
   // It didn't work.
      MessageBox("Error", &
         "The delete failed. Employee ID "&
         +"is not valid.")
         end if
   else
   // No input value. Prompt user.
      MessageBox("Error", &
          "An employee ID is required for "&
         +"delete!")
   end if
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 3

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. The user wants to extract rows from the employee table and insert them into the table named extract_employees. The extraction occurs when the user clicks the button Cb_Extract. The boolean variable YoungWorkers is set to TRUE or FALSE elsewhere in the application.

The script for the Clicked event for the CommandButton Cb_Extract is:

```
integer   EmployeeAgeLowerLimit
integer   mployeeAgeUpperLimit
// Do they have young workers?
if (YoungWorkers = TRUE) then
// Yes - set the age limit in the YOUNG range.
// Assume no employee is under legal working age.
   EmployeeAgeLowerLimit = 16
// Pick an upper limit.
   EmployeeAgeUpperLimit = 42
else
// No - set the age limit in the OLDER range.
   EmployeeAgeLowerLimit = 43
// Pick an upper limit that includes all employees.
   EmployeeAgeUpperLimit = 200
```

```
end if

INSERT INTO extract_employees(emp_id,emp_name)
   SELECT emp_id, emp_name FROM employee
      WHERE emp_age >= :EmployeeAgeLowerLimit
      AND emp_age <= :EmployeeAgeUpperLimit;
```

ODBC Cursor statements

In embedded SQL, statements that retrieve data and statements that update data can both involve cursors. Not all backend DBMSs support cursor statements.

Retrieval statements

The retrieval statements that involve cursors are:

- DECLARE *cursor_name* CURSOR FOR ...
- OPEN *cursor_name*
- FETCH *cursor_name* INTO ...
- CLOSE *cursor_name*

Update statements

The update statements that involve cursors are:

- UPDATE ... WHERE CURRENT OF *cursor_name*
- DELETE ... WHERE CURRENT OF *cursor_name*

See also

ODBC Retrieval using cursors
ODBC FETCH NEXT
ODBC FETCH FIRST, FETCH PRIOR, and FETCH LAST
ODBC Update

ODBC Retrieval using cursors

Retrieval using cursors is conceptually similar to the singleton SELECT discussed earlier. The main difference is that since there can be multiple rows in a result set, you control when the next row is fetched into PowerScript variables.

For example, if you expect only a single row to exist in the employee table for each emp_id, use a singleton SELECT statement. In a singleton SELECT, you specify the SELECT statement and destination variables in one concise SQL statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

However, if the SELECT may return multiple rows, you must:

1 Declare a cursor.

2 Open it (which conceptually executes the SELECT).

3 Fetch rows as needed.

4 Close the cursor.

Declaring and opening a cursor

Declaring a cursor is tightly coupled with the OPEN statement. The DECLARE specifies the SELECT statement to be executed, and the OPEN actually executes it.

Declaring a cursor is similar to declaring a variable. A cursor declaration is a nonexecutable statement just like a variable declaration. The first step in declaring a cursor is to define how the result set looks. To do this, you need a SELECT statement, and since you must refer to the result set in subsequent SQL statements, you must associate the result set with a logical name.

Example

Assume the SingleLineEdit sle_1 contains the state code for the retrieval:

```
// Declare cursor emp_curs for employee table
// retrieval.
DECLARE emp_curs CURSOR FOR
    SELECT emp_id, emp_name FROM EMPLOYEE
    WHERE emp_state = :sle_1.text;

// Declare local variables for retrieval.
string emp_id_var
string emp_name_var

// Execute the SELECT statement with
// the current value of sle_1.text.
OPEN emp_curs;

// At this point, if there are no errors,
// the cursor is available for further
// processing.
```

Scrolling and locking

Use the DBParm parameters CursorScroll and CursorLock to specify the scrolling and locking options.

Note Not all DBMSs support these scrolling and locking options.

Fetching rows

The ODBC interface supports the following FETCH statements. You can use them if they are supported by your backend DBMS.

- FETCH NEXT
- FETCH FIRST
- FETCH PRIOR
- FETCH LAST

Note Not all DBMSs support all of these FETCH statements.

ODBC FETCH NEXT

In the singleton SELECT, you specify variables to hold values for the columns within the selected row. The syntax of the FETCH statement is similar to the singleton SELECT statement syntax. Values are returned INTO a specified list of variables.

Example

This example continues the previous example by retrieving some data:

```
// Go get the first row from the result set.
FETCH emp_curs INTO :emp_id_var, :emp_name_var;
```

If at least one row is retrieved, this FETCH places the values of the emp_id and emp_name columns from the first row in the result set into the PowerScript variables emp_id_var and emp_name_var. FETCH statements typically occur in a loop that processes several rows from a result set (one row at a time), but this is not the only way they are used.

What happens when the result set is exhausted?
FETCH returns +100 (not found) in the SQLCode property within the referenced transaction object. This is an informational return code; -1 in SQLCode indicates an error.

See also

ODBC FETCH FIRST, FETCH PRIOR, and FETCH LAST

ODBC FETCH FIRST, FETCH PRIOR, and FETCH LAST

In addition to the conventional FETCH NEXT statement, the ODBC interface supports FETCH FIRST, FETCH PRIOR, and FETCH LAST statements.

What happens if you only enter FETCH?
If you only enter FETCH, PowerBuilder assumes FETCH NEXT.

Closing the cursor

The CLOSE statement terminates processing for the specified cursor. CLOSE releases resources associated with the cursor, and subsequent references to that cursor are allowed only if another OPEN is executed. Although you can have multiple cursors open at the same time, you should close the cursors as soon as possible for efficiency reasons.

See also

ODBC FETCH NEXT

ODBC Update

After a FETCH statement completes successfully, you are positioned on a current row within the cursor. At this point, you can execute an UPDATE or DELETE statement using the WHERE CURRENT OF cursor_name syntax to update or delete the row. PowerBuilder enforces the cursor update restrictions of the backend database, and violations will result in an execution error.

Example

This cursor example illustrates how to loop through a result set. It assumes that the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed. The statements retrieve rows from the employee table, and then display a message box with the employee name for each row that is found.

```
// Declare the emp_curs.
DECLARE emp_curs CURSOR FOR
    SELECT emp_name FROM EMPLOYEE
    WHERE emp_state = :sle_1.text;

// Declare a destination variable for employee
// names.
string emp_name_var

// Execute the SELECT statement with the
// current value of sle_1.text.
OPEN emp_curs;
```

```
// Fetch the first row from the result set.
FETCH emp_curs INTO :emp_name_var;
// Loop through result set until exhausted.
DO WHILE sqlca.sqlcode = 0
// Display a message box with the employee name.
   MessageBox("Found an employee!",emp_name_var)
// Fetch the next row from the result set.
   FETCH emp_curs INTO :emp_name_var;
LOOP
// All done; close the cursor.
CLOSE emp_curs;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

ODBC Database stored procedures

Retrieval and update

You can use database stored procedures for:

- Retrieval only
- Update only
- Retrieval and update

Your DBMS

Not all DBMSs support these retrieval and update options.

Using stored procedures

When you use database stored procedures in a PowerBuilder application, keep the following points in mind:

- **Manipulating stored procedures** PowerBuilder provides SQL statements that are similar to cursor statements for manipulating database stored procedures.
- **Retrieval and update** PowerBuilder supports retrieval, update, or a combination of retrieval and update in database stored procedures, including procedures that do not return a result set and those that return a result set.

- **Transactions and stored procedures without result sets** When a procedure executes using a particular connection (transaction) and the procedure does not return a result set, the procedure is no longer active. No result set is pending, and therefore you do not execute a CLOSE statement.

See also

ODBC Retrieval
ODBC Using database stored procedures in DataWindow objects

ODBC Retrieval

PowerBuilder uses a construct similar to cursors to support retrieval using database stored procedures. PowerBuilder supports four embedded SQL statements that involve database stored procedures:

- DECLARE *procedure_name* PROCEDURE FOR ...
- EXECUTE *procedure_name*
- FETCH *procedure_name* INTO ...
- CLOSE *procedure_name*

See also

ODBC DECLARE and EXECUTE
ODBC EXECUTE
ODBC FETCH
ODBC CLOSE

ODBC DECLARE and EXECUTE

PowerBuilder requires a declarative statement to identify the database stored procedure that is being used and to specify a logical name for the procedure. The logical name is used to reference the procedure in subsequent SQL statements.

The general syntax for declaring a procedure is:

```
DECLARE logical_procedure_name PROCEDURE FOR
    procedure_name
    {@param1 = value, @param2 = value2, ...}
    {USING transaction_object};
```

where *logical_procedure_name* can be any valid PowerScript identifier and *procedure_name* is the name of a stored procedure in the database.

The parameter references can take the form of any valid parameter string the database accepts. PowerBuilder inspects the parameter list format only for variable substitution. The USING clause is required only if you are using a transaction object other than the default transaction object (SQLCA).

Output parameters might not be returned when you use an embedded SQL command to call a stored procedure. You can set the PBNewSPInvocation database parameter to "Yes" to use an alternative method to invoke a stored procedure. The behavior of the PowerBuilder ODBC driver when this DBParm is set is consistent with the default behavior of the OLE DB and JDBC drivers.

If PBNewSPInvocation is set to "Yes," the alternative method is used when you retrieve data into a DataWindow object that uses a stored procedure. See ODBC DECLARE and EXECUTE with PBNewSPInvocation

Example

Assume a stored procedure named proc1 is defined on the server. To declare proc1 for processing within PowerBuilder, enter:

```
DECLARE emp_proc PROCEDURE FOR proc1;
```

The procedure declaration is a nonexecutable statement, just like a cursor declaration. However, where cursors have an OPEN statement, procedures have an EXECUTE statement.

When an EXECUTE statement executes, the procedure is invoked. The EXECUTE refers to the logical procedure name, in this example emp_proc:

```
EXECUTE emp_proc;
```

See also

ODBC EXECUTE
ODBC DECLARE and EXECUTE with PBNewSPInvocation

ODBC DECLARE and EXECUTE with PBNewSPInvocation

PowerBuilder requires a declarative statement to identify the database stored procedure that is being used and to specify a logical name for the procedure. The logical name is used to reference the procedure in subsequent SQL statements.

The general syntax for declaring a procedure is:

```
DECLARE logical_procedure_name PROCEDURE FOR
    procedure_name
    @param1 = value, @param2 = value2,
    @PARAM3 = VALUE3 OUTPUT
{USING transaction_object};
```

where *logical_procedure_name* can be any valid PowerScript identifier and *procedure_name* is the name of a stored procedure in the database. Use the OUT or OUTPUT keyword to obtain the value of the output parameter.

The parameter references can take the form of any valid parameter string the database accepts. PowerBuilder inspects the parameter list format only for variable substitution. The USING clause is required only if you are using a transaction object other than the default transaction object (SQLCA).

You must set the PBNewSPInvocation database parameter to ‘Yes’ to use this method to invoke a stored procedure. The behavior of the PowerBuilder ODBC driver when this DBParm is set is consistent with the default behavior of the OLE DB and JDBC drivers.

If PBNewSPInvocation is set to ‘Yes’, this method is used when you retrieve data into a DataWindow object that uses a stored procedure. This DBParm has no effect when you use RPC to invoke a stored procedure.

If PBNewSPInvocation is set to ‘No’, use the syntax described in ODBC DECLARE and EXECUTE.

Example 1

Assume a stored procedure named proc1 is defined on the server as:

```
CREATE PROCEDURE proc1 AS
    SELECT emp_name FROM employee
```

To declare proc1 for processing within PowerBuilder, enter:

```
DECLARE emp_proc PROCEDURE FOR proc1;
```

The procedure declaration is a nonexecutable statement, just like a cursor declaration. However, where cursors have an OPEN statement, procedures have an EXECUTE statement.

When an EXECUTE statement executes, the procedure is invoked. The EXECUTE refers to the logical procedure name, in this example emp_proc:

```
EXECUTE emp_proc;
```

Example 2

To declare a procedure with input and output parameters, enter:

```
DECLARE sp_duration PROCEDURE FOR pr_date_diff_prd_ken
    @var_date_1 = :ad_start,
    @var_date_2 = :ad_end,
    @rtn_diff_prd = :ls_duration OUTPUT;
```

If the stored procedure contains result sets, you must fetch the result sets first. If the stored procedure has a return value and you want to obtain it, use the format RC=*procedure_name*:

```
DECLARE sp_duration PROCEDURE FOR
RC=pr_date_diff_prd_ken
    @var_date_1 = :ad_start,
    @var_date_2 = :ad_end,
    @rtn_diff_prd = :ls_duration OUTPUT;
```

ODBC FETCH

To access rows returned in a result set, use the FETCH statement the same way you use it for cursors. The FETCH statement can be executed after any successful EXECUTE statement for a procedure that returns a result set.

Example

```
FETCH emp_proc INTO :emp_name_var;
```

Using FETCH after EXECUTE
Following an EXECUTE statement for a procedure, you can use the FETCH statement only to access values produced by the SELECT statement in the database stored procedure.

Since PowerBuilder cannot determine at compile time what result set will be returned when a database stored procedure executes, you must code FETCH statements so that the stored procedure exactly matches the format of the result set during execution. Assume you coded the second FETCH statement in the example above as:

```
FETCH emp_proc INTO :var1, :var2, :var3;
```

The statement compiles without errors. However, you will get an execution error indicating that the number of columns in the FETCH statement does not match the number of columns in the result set.

See also

ODBC EXECUTE
ODBC FETCH NEXT
ODBC FETCH FIRST, FETCH PRIOR, and FETCH LAST

ODBC CLOSE

If a database stored procedure returns a result set, you must close the stored procedure when processing is complete. The procedure remains open until you close it, execute a COMMIT or ROLLBACK, or end the database connection.

Do you have to retrieve all the rows?
You do not have to retrieve all rows in a result set to close a request or procedure.

Example

Closing a procedure looks the same as closing a cursor:

```
CLOSE emp_proc;
```

ODBC EXECUTE

Database stored procedures that perform only updates and do not return a result set are handled in much the same way as procedures that return a result set. The only difference is that after the EXECUTE procedure_name statement executes, no result set is pending, so a CLOSE statement is not required.

Using the SQLCode property

If a specific procedure can never return a result set, only the EXECUTE statement is required. If a procedure may or may not return a result set, you can test the SQLCode property of the referenced transaction object for +100 (the code for NOT FOUND) after the EXECUTE.

The possible values for SQLCode after an EXECUTE are:

Return code	Means
0	The EXECUTE was successful and a result set is pending. Regardless of the number of FETCH statements executed, the procedure must be explicitly closed with a CLOSE statement. Fetched row not found.
+100	Fetched row not found.
-1	The EXECUTE was not successful and no result set was returned.

Example 1

This example illustrates how to execute a stored procedure that does not return a result set. It assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed.

```
// good_employee is a database stored procedure.
// Declare the procedure.
DECLARE good_emp_proc PROCEDURE
    FOR good_employee;

// Execute it.
EXECUTE good_emp_proc;

// Test return code. Allow for +100 since you
// do not expect a result set.
if SQLCA.sqlcode = -1 then

// Issue an error message since it failed.
    MessageBox("Stored Procedure Error!", &
    SQLCA.sqlerrtext)
end if
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

This example illustrates how to pass parameters to a database stored procedure. It assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed. Emp_id_var was set to 691 elsewhere.

```
// get_employee is a database stored procedure.
// Declare the procedure.
DECLARE get_emp_proc PROCEDURE FOR
    get_employee @emp_id_parm = :emp_id_var;

// Declare a destination variable for emp_name.
string emp_name_var

// Execute the stored procedure using the
// current value for emp_id_var.
EXECUTE get_emp_proc;

// Test return code to see if it worked.
if SQLCA.sqlcode = 0 then

// Since we got a row, fetch it and display it.
    FETCH get_emp_proc INTO :emp_name_var;

// Display the employee name.
    MessageBox("Got my employee!",emp_name_var)

// You are all done, so close the procedure.
    CLOSE Get_emp_proc;
```

```
end if
```

ODBC Using database stored procedures in DataWindow objects

You can use database stored procedures as a data source for DataWindow objects. The following rules apply:

- **Result set definition** You must define what the result set looks like in the DataWindow painter. PowerBuilder cannot determine this information from the stored procedure definition in the database.
- **Stored procedure arguments** The DataWindow painter provides the arguments for stored procedures only if the ODBC driver you are using to connect gives PowerBuilder the required information. If the arguments for the database stored procedure are not provided, you must define them.
- **DataWindow updates** Updates are not allowed for stored procedures in a DataWindow object. Only retrieval is allowed.
- **ODBC syntax** PowerBuilder supports the syntax appropriate for all backend databases supported by the ODBC interface. In the DataWindow painter, PowerBuilder displays the most general stored procedure syntax. It then converts it to the syntax appropriate for the backend database before passing it to the database.

CHAPTER 17 Using Embedded SQL with JDBC

When you use the JDBC interface to connect to a database, you can use embedded SQL in your scripts.

JDBC DECLARE and EXECUTE

PowerBuilder requires a declarative statement to identify the database stored procedure that is being used and a logical name that can be referenced in subsequent SQL statements.

The general syntax for declaring a procedure is:

```
DECLARE logical_procedure_name PROCEDURE FOR
    procedure_name
    @Param1 = value1, @Param2 = value2 ,
    @Param3 = value3 OUTPUT,
    {USING transaction_object} ;
```

where *logical_procedure_name* can be any valid PowerScript data identifier and *procedure_name* is the name of the stored procedure in the database.

The parameter references can take the form of any valid parameter string that JDBC accepts. PowerBuilder does not inspect the parameter list format except for purposes of variable substitution. You must use the reserved word OUTPUT to indicate an output parameter. The USING clause is required only if you are using a transaction object other than the default transaction object (SQLCA).

Example 1

Assume a stored procedure proc1 is defined as:

```
CREATE PROCEDURE proc1 AS
    SELECT emp_name FROM employee
```

To declare that procedure for processing within PowerBuilder, enter:

```
DECLARE emp_proc PROCEDURE FOR proc1;
```

Note that this declaration is a nonexecutable statement, just like a cursor declaration. Where cursors have an OPEN statement, procedures have an EXECUTE statement.

When an EXECUTE statement executes, the procedure is invoked. The EXECUTE refers to the logical procedure name:

```
EXECUTE emp_proc;
```

Example 2

To declare a procedure with input and output parameters, enter:

```
DECLARE sp_duration PROCEDURE FOR pr_date_diff_prd_ken
    @var_date_1 = :ad_start,
    @var_date_2 = :ad_end,
    @rtn_diff_prd = :ls_duration OUTPUT;
```

CHAPTER 18

Using Embedded SQL with OLE DB

About this chapter

When you create scripts for a PowerBuilder application, you can use embedded SQL statements in the script to perform operations on the database. The features supported when you use embedded SQL depend on the DBMS to which your application connects.

Overview

When you use the PowerBuilder OLE DB interface to connect to a backend database, you can use embedded SQL in your scripts.

You can embed the following types of SQL statements in scripts and user-defined functions if the OLE DB driver you are using and the backend DBMS you are accessing supports this functionality. (Not all backend databases support cursor statements and database stored procedures.)

- Transaction management statements
- Non-cursor statements
- Cursor statements
- Database stored procedures

OLE DB Programming Models

OLE DB is a set of COM (Component Object Model) interfaces that provide uniform access to data stored in multiple, diverse data sources. These data sources also enable applications to provide additional database services.

When you use embedded SQL, PowerBuilder makes the required calls to the backend database. Therefore, you do not need to know anything about the OLE DB interface to use embedded SQL with PowerBuilder.

See also

Chapter 4, Using the OLE DB Interface
OLE DB SQL support
OLE DB Transaction management statements
OLE DB Non-cursor statements
OLE DB Cursor statements
OLE DB Database stored procedures

OLE DB SQL support

PowerBuilder embedded SQL supports the name qualification conventions and functions used in the databases accessible through the PowerBuilder OLE DB interface.

See also

OLE DB Name qualification
OLE DB SQL functions

OLE DB Name qualification

PowerBuilder does not inspect all SQL statement syntax, so you can qualify database catalog entities as necessary.

For example, the following qualifications are acceptable for a PowerBuilder OLE DB interface to a SQL Anywhere database:

- emp_name
- employee.emp_name

OLE DB SQL functions

In SQL statements, you can use any function that your backend DBMS supports (such as aggregate or mathematical functions). For example, if your DBMS supports the function Sum, you can use the function Sum in a SELECT statement:

```
SELECT Sum(salary)
   INTO :salary_sum_var
   FROM employee;
```

Calling OLE DB functions

While PowerBuilder provides access to a large percentage of the features within OLE DB, in some cases you might decide that you need to call one or more OLE DB functions directly for a particular application. PowerBuilder provides access to most Windows DLLs by using external function declarations.

PowerBuilder OLE DB can export OLE DB data source objects or session objects to users using the PowerScript function DBHandle. Users can create their own session objects using the exported data source object, so they can get a new independent connection that has connection properties similar to those used by PowerBuilder OLE DB. With the exported session object, users can also create their own command object that is under PowerBuilder OLE DB's transaction scope. The behavior is like using DBHandle() with the PowerBuilder ODBC interface.

DBHandle

DBHandle takes a transaction object as a parameter and returns a long variable, which is an interface pointer to a data source object or a session object. By default PowerBuilder OLE DB exports a data source object. If the DBParm "ReturnCommandHandle=1" is set, PowerBuilder OLE DB exports a session object.

Example 1

This example illustrates how to use DBHandle to get an OLE DB data source object. As with other examples, assume a successful connection has occurred using the default transaction object (SQLCA).

```
// Define a variable to hold the DB connection handle.
Long OleDbCnnInterface

// Get OLE DB Data Source Object
OleDbCnnInterface = SQLCA.DBHandle()

// Now that you have the OLE DB data source object,
// call the DLL function.
MyDLLFunction(OleDbCnnInterface, parm1, parm2)

// In your DLL, cast the incoming handle to the
// IUnknown* interface

MyDLLFunction(long OleDbCnnInterface,
                 parm1_type parm1,
                 parm2_type Parm2, ...)
{
    IUnknown* pUnkDataSource = &
        (IUnknown*)OleDbCnnInterface;
    IDBCreateSession*  pIDBCreateSession = NULL;

    pUnkDataSource-
>QueryInterface(IID_IDBCreateSession,
    (void**)&pIDBCreateSession));

// now you have the OLE DB IDBCreateSession interface,
// you can create your own independent session object
// from the PowerBuilder OLE DB driver
    IUnknown **  ppUnkSession;
    pIDBCreateSession->CreateSession(NULL, //pUnkOuter
                        IID_IDBCreateCommand, //riid
                        ppUnkSession          //ppSession
                        );

}
```

Example 2

This example illustrates how to use DBHandle to get an OLE DB session object.

```
// Before connection, set DBParm ReturnCommandHandle=1
...
SQLCA.DBParm = "ReturnCommandHandle = 1"
CONNECT;

// After successful connection
// Define a variable to hold the DB connection handle.
long    OleDbCnnInterface

// Get OLE DB session object
OleDbCnnInterface = SQLCA.DBHandle()

// Now you have the OLE DB session object,
// call the DLL function.
MyDLLFunction(OleDbCnnInterface, parm1, parm2)

// In your DLL, cast the incoming handle to
// IUnknown* interface

MyDLLFunction(long OleDbCnnInterface,
    parm1_type parm1,
    parm2_type Parm2, ...)

{
        IUnknown* pUnkSession = &
            (IUnknown*)OleDbCnnInterface;
            IDBCreateCommand *  pIDBCreateCommand = NULL;

            pUnkSession->QueryInterface &
                    (IID_IDBCreateCommand,
                (void**)&pIDBCreateCommand));
```

With the IDBCreateCommand interface used by the PowerBuilder OLE DB interface, you can create your own command object. Your command object and the PowerBuilder command object will be in the same transaction scope.

See also

OLE DB Using ODBC escape Sequences

OLE DB Using ODBC escape Sequences

ODBC defines extensions that are common to most backend DBMSs. To cover vendor-specific extensions, the syntax defined by ODBC uses the escape clause provided by the X/Open and SQL Access Group (SAG) SQL draft specifications. OLE DB supports ODBC escape sequences directly.

For example, some of the extensions defined in ODBC are:

- Date, Time, and Timestamp Literals
- Scalar functions (such as data type, numeric, and string conversion functions)
- Outer joins
- Procedure Calls

Note For maximum portability, you should use escape sequences in your applications.

Syntax

For example, PowerBuilder uses the date, time, and timestamp escape clauses as the default formats for data manipulation. The syntax for each of these escape clauses is:

```
{ d yyyy-mm-dd }
```

```
{ t hh:mm:ss }
```

```
{ ts yyyy-mm-dd hh:mm:ss:[fff[fff]] }
```

Example

Each of the following statements updates employee Henry Jones's start time in the Employee table. The first statement uses the escape clause, and the second statement uses native syntax for a time column:

```
UPDATE Employee
    SET start_time = {t 08:30:00}
    WHERE emp_name = "Henry Jones"
```

```
UPDATE Employee
    SET start_time = (08:30:00)
    WHERE emp_name = "Henry Jones"
```

OLE DB Transaction management statements

If the database you are connecting to supports transaction management, you can use the following transaction management statements with one or more transaction objects to manage connections and transactions for a database:

- CONNECT
- DISCONNECT
- COMMIT
- ROLLBACK

See also OLE DB Using CONNECT, DISCONNECT, COMMIT, and ROLLBACK

OLE DB Using CONNECT, DISCONNECT, COMMIT, and ROLLBACK

The following table lists each transaction management statement and describes how it works when you use the PowerBuilder OLE DB interface to connect to a database:

Statement	Description
CONNECT	Establishes the database connection. After you assign values to the required properties of the transaction object, you can execute a CONNECT. After the CONNECT completes successfully, by default PowerBuilder automatically starts a transaction. Set SQLCA.AutoCommit=TRUE to tell PowerBuilder not to start a transaction automatically.
DISCONNECT	Terminates a successful connection. When a DISCONNECT is executed, PowerBuilder internally executes a COMMIT WORK statement to commit all changes and then issues a CLOSE DATABASE statement to terminate the logical unit of work.
COMMIT	Applies all changes made to the database since the beginning of the current unit of work.
ROLLBACK	Undoes all changes made to the database since the beginning of the current logical unit of work.

See also OLE DB Performance and locking

OLE DB Performance and locking

After a connection is established, SQL statements can cause locks to be placed on database entities. The more locks there are in place at a given moment in time, the more likely it is that the locks will hold up another transaction.

Rules

No set of rules for designing a database application is totally comprehensive. However, when you design a PowerBuilder application, you should do the following:

- **Long-running connections** Determine whether you can afford to have long-running connections. If not, your application should connect to the database only when absolutely necessary. After all the work for that connection is complete, the transaction should be disconnected.

  If long-running connections are acceptable, then COMMITs should be issued as often as possible to guarantee that all changes do in fact occur. More importantly, COMMITs should be issued to release any locks that may have been placed on database entities as a result of the statements executed using the connection.

- **SetTrans or SetTransObject function** Determine whether you want to use default DataWindow transaction processing (the SetTrans function) or control the transaction in a script (the SetTransObject function).

  If you cannot afford to have long-running connections and therefore have many short-lived transactions, use the default DataWindow transaction processing. If you want to keep connections open and issue periodic COMMITs, use the SetTransObject function and control the transaction yourself.

Switching during a connection
To switch between transaction processing and AutoCommit during a connection, change the setting of AutoCommit in the transaction object.

Isolation feature

OLE DB uses the isolation feature to support assorted database lock options. In PowerBuilder, you can use the Lock property of the transaction object to set the isolation level when you connect to the database.

The following example shows how to set the Lock property to RU (Read uncommitted):

```
// Set the lock property to read uncommitted
// in the default transaction object SQLCA.
SQLCA.Lock = "RU"
```

Example 1

This script uses embedded SQL to connect to a database and attempts to insert a row in the ORDER_HEADER table and a row in the ORDER_ITEM table. The script then executes a COMMIT or ROLLBACK depending on the success of all statements in the script.

```
// Set the SQLCA connection properties.
SQLCA.DBMS = "OLE DB"
SQLCA.DBParm = "PROVIDER='SAOLEDB.10',DATASOURCE= &
    'SQL Anywhere 10 Demo'”

// Connect to the database.
CONNECT USING SQLCA;

// Insert a row into the ORDER_HEADER table.
INSERT INTO ORDER_HEADER (ORDER_ID,CUSTOMER_ID)
    VALUES (7891, 129);

// Test return code for ORDER_HEADER insertion.
// A ROLLBACK is required only if the first row
// was inserted successfully.
if SQLCA.sqlcode = 0 then

// Since the ORDER_HEADER is inserted,
// try to insert ORDER_ITEM.
    INSERT INTO ORDER_ITEM
        (ORDER_ID, ITEM_NBR, PART_NBR, QTY)
        VALUES (7891, 1, '991PLS', 456);
// Test return code for ORDER_ITEM insertion.
    if SQLCA.sqlcode = -1 then

// If insert failed, ROLLBACK insertion of
// ORDER_HEADER.
    ROLLBACK USING SQLCA;
    end if
end if

// Disconnect from the database.
DISCONNECT USING SQLCA;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

This example uses scripts for the Open and Close events for a window and the Clicked event for a CommandButton to illustrate how you can manage transactions for a DataWindow control. Assume a window contains a DataWindow control dw_1 and a CommandButton Cb_Update. Also assume the user enters data in dw_1 and then clicks the Cb_Update button to update the database with the data.

The window OPEN event script:

```
// Set the transaction object properties
// and connect to the database.
// Set the SQLCA connection properties.
SQLCA.DBMS = "OLE DB"
SQLCA.DBParm = "PROVIDER='SAOLEDB.10',DATASOURCE= &
    'SQL Anywhere 10 Demo'”

// Connect to the database.
CONNECT USING SQLCA;

// Tell the DataWindow which transaction object
// to use.
dw_1.SetTransObject(sqlca)
```

The CommandButton CLICKED event script:

```
// Declare ReturnValue an integer.
integer    ReturnValue
ReturnValue = dw_1.Update( )

// Test to see if updates were successful.
if ReturnValue = -1 then

// Updates were not successful. Since we used
// SetTransObject, roll back any changes made
// to the database.
    ROLLBACK USING SQLCA;
else

// Updates were successful. Since we used
// SetTransObject, commit any changes made
// to the database.
    COMMIT USING SQLCA;
end if
```

The window CLOSE event script:

```
// Disconnect from the database.
DISCONNECT USING SQLCA;
```

OLE DB Non-cursor statements

The statements that do not involve cursors are:

- DELETE
- INSERT
- UPDATE
- SELECT

OLE DB DELETE, INSERT, and UPDATE

Internally, PowerBuilder processes DELETE, INSERT, and UPDATE the same way. PowerBuilder inspects these statements for variable references and replaces all variable references with a constant that conforms to the backend database's rules for that data type.

Example

Assume you enter the following statement:

```
DELETE FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var is a PowerScript variable with the data type of integer that has been defined within the scope of the script that contains the DELETE statement.

Before the DELETE statement is executed, emp_id_var is assigned a value (say 691) so when the DELETE statement executes, the database receives the following command:

```
DELETE FROM employee WHERE emp_id = 691;
```

When is this substitution technique used?
This variable substitution technique is used for all PowerScript variable types. When you use embedded SQL, precede all PowerScript variables with a colon (:).

See also

OLE DB SELECT

OLE DB SELECT

The SELECT statement contains input and output variables.

- Input variables are passed to the database as part of the execution, and the substitution is as described for DELETE, INSERT, and UPDATE.
- Output variables return values based on the result of the SELECT statement.

Example 1

Assume you enter the following statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var, emp_salary_var, and emp_name_var are PowerScript variables defined within the scope of the script containing the SELECT statement, and emp_id_var is an input variable and is processed as described in the DELETE example above.

Both emp_name_var and emp_salary_var are output variables that will be used to return values from the database. The data types of emp_name_var and emp_salary_var should be the PowerScript data types that best match the data type in the database. When the data types do not match perfectly, PowerBuilder converts them.

How big should numeric output variables be?
For numeric data, the output variable must be large enough to hold any value that may come from the database.

Assume the value for emp_id_var is 691 as in the previous example. When the SELECT statement executes, the database receives this command:

```
SELECT emp_name, emp_salary
    FROM employee WHERE emp_id = 691;
```

If no errors are returned when the statement executes, data locations are bound internally for the result fields. The data returned into these locations is converted if necessary, and the appropriate PowerScript variables are set to those values.

Example 2

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. It also assumes the data type of the emp_id column in the employee table is CHARACTER[10]. The user enters an employee ID into the single line edit field sle_Emp and clicks the button Cb_Delete.

The script for the Clicked event in the CommandButton Cb_Delete is:

```
// Make sure we have a value.
if sle_Emp.text <> "" then

// Since we have a value, let's try to
// delete it.
    DELETE FROM employee
    WHERE emp_id = :sle_Emp.text;

// Test to see if the DELETE worked.
    if SQLCA.sqlcode = 0 then

// It seems to have worked; let user know.
    MessageBox("Delete",&
        "The delete has been successfully "&
        +"processed!")
    else

// It didn't work.
    MessageBox("Error", &
        "The delete failed. Employee ID "&
        +"is not valid.")
        end if
else

// No input value. Prompt user.
    MessageBox("Error", &
         "An employee ID is required for "&
        +"delete!")
end if
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 3

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. The user wants to extract rows from the employee table and insert them into the table named extract_employees. The extraction occurs when the user clicks the button Cb_Extract. The boolean variable YoungWorkers is set to TRUE or FALSE elsewhere in the application.

The script for the Clicked event for the CommandButton Cb_Extract is:

```
integer    EmployeeAgeLowerLimit
integer    EmployeeAgeUpperLimit
```

```
// Do they have young workers?
if (YoungWorkers = TRUE) then

// Yes - set the age limit in the YOUNG range.
// Assume no employee is under legal working age.
   EmployeeAgeLowerLimit = 16

// Pick an upper limit.
   EmployeeAgeUpperLimit = 42
else

// No - set the age limit in the OLDER range.
   EmployeeAgeLowerLimit = 43

// Pick an upper limit that includes all employees.
   EmployeeAgeUpperLimit = 200
end if

INSERT INTO extract_employees(emp_id,emp_name)
   SELECT emp_id, emp_name FROM employee
      WHERE emp_age >= :EmployeeAgeLowerLimit
      AND emp_age <= :EmployeeAgeUpperLimit;
```

OLE DB Cursor statements

In embedded SQL, statements that retrieve data can involve cursors. PowerBuilder OLE DB supports only forward, read-only cursors.

Retrieval statements

The retrieval statements that involve cursors are:

- DECLARE *cursor_name* CURSOR FOR ...
- OPEN *cursor_name*
- FETCH *cursor_name* INTO ...
- CLOSE *cursor_name*

Update statements

`UPDATE ... WHERE CURRENT OF` ***`cursor_name`*** and `DELETE ... WHERE CURRENT OF` ***`cursor_name`*** are not supported.

See also

OLE DB Retrieval using cursors
OLE DB FETCH NEXT

OLE DB Retrieval using cursors

Retrieval using cursors is conceptually similar to the singleton SELECT discussed earlier. The main difference is that since there can be multiple rows in a result set, you control when the next row is fetched into PowerScript variables.

For example, if you expect only a single row to exist in the employee table for each emp_id, use a singleton SELECT statement. In a singleton SELECT, you specify the SELECT statement and destination variables in one concise SQL statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

However, if the SELECT may return multiple rows, you must:

1 Declare a cursor.

2 Open it (which conceptually executes the SELECT).

3 Fetch rows as needed.

4 Close the cursor.

Declaring and opening a cursor

Declaring a cursor is tightly coupled with the OPEN statement. The DECLARE specifies the SELECT statement to be executed, and the OPEN actually executes it.

Declaring a cursor is similar to declaring a variable. A cursor declaration is a nonexecutable statement just like a variable declaration. The first step in declaring a cursor is to define how the result set looks. To do this, you need a SELECT statement, and since you must refer to the result set in subsequent SQL statements, you must associate the result set with a logical name.

Example

Assume the SingleLineEdit sle_1 contains the state code for the retrieval:

```
// Declare cursor emp_curs for employee table
// retrieval.
DECLARE emp_curs CURSOR FOR
    SELECT emp_id, emp_name FROM EMPLOYEE
    WHERE emp_state = :sle_1.text;

// Declare local variables for retrieval.
string emp_id_var
string emp_name_var

// Execute the SELECT statement with
// the current value of sle_1.text.
```

```
OPEN emp_curs;

// At this point, if there are no errors,
// the cursor is available for further
// processing.
```

Fetching rows The PowerBuilder OLE DB interface supports FETCH statements.

See also OLE DB FETCH NEXT

OLE DB FETCH NEXT

In the singleton SELECT, you specify variables to hold values for the columns within the selected row. The syntax of the FETCH statement is similar to the singleton SELECT statement syntax. Values are returned INTO a specified list of variables.

Example This example continues the previous example by retrieving some data:

```
// Go get the first row from the result set.
FETCH emp_curs INTO :emp_id_var, :emp_name_var;
```

If at least one row is retrieved, this FETCH places the values of the emp_id and emp_name columns from the first row in the result set into the PowerScript variables emp_id_var and emp_name_var. FETCH statements typically occur in a loop that processes several rows from a result set (one row at a time), but this is not the only way they are used.

What happens when the result set is exhausted?
FETCH returns +100 (not found) in the SQLCode property within the referenced transaction object. This is an informational return code; -1 in SQLCode indicates an error.

Closing the cursor The CLOSE statement terminates processing for the specified cursor. CLOSE releases resources associated with the cursor, and subsequent references to that cursor are allowed only if another OPEN is executed. Although you can have multiple cursors open at the same time, you should close the cursors as soon as possible for efficiency reasons.

OLE DB Database stored procedures

Retrieval and update

You can use database stored procedures for:

- Retrieval only
- Update only
- Retrieval and update

Your DBMS

Not all DBMSs support these retrieval and update options.

Using stored procedures

When you use database stored procedures in a PowerBuilder application, keep the following points in mind:

- **Manipulating stored procedures** PowerBuilder provides SQL statements that are similar to cursor statements for manipulating database stored procedures.
- **Retrieval and update** PowerBuilder supports retrieval, update, or a combination of retrieval and update in database stored procedures, including procedures that do not return a result set and those that return a result set.
- **Transactions and stored procedures without result sets** When a procedure executes using a particular connection (transaction) and the procedure does not return a result set, the procedure is no longer active. No result set is pending, and therefore you do not execute a CLOSE statement.

See also

OLE DB Retrieval
OLE DB Using database stored procedures in DataWindow objects

OLE DB Retrieval

PowerBuilder uses a construct similar to cursors to support retrieval using database stored procedures. PowerBuilder supports four embedded SQL statements that involve database stored procedures:

- DECLARE *procedure_name* PROCEDURE FOR ...
- EXECUTE *procedure_name*
- FETCH *procedure_name* INTO ...
- CLOSE *procedure_name*

See also

OLE DB DECLARE and EXECUTE

OLE DB EXECUTE
OLE DB FETCH
OLE DB CLOSE

OLE DB DECLARE and EXECUTE

PowerBuilder requires a declarative statement to identify the database stored procedure that is being used and to specify a logical name for the procedure. The logical name is used to reference the procedure in subsequent SQL statements.

The general syntax for declaring a procedure is:

```
DECLARE logical_procedure_name PROCEDURE FOR
    [RC=]procedure_name
    {@param1 = value [OUTPUT], @param2 = &
       value2[OUTPUT], ...}
    {USING transaction_object};
```

where *logical_procedure_name* can be any valid PowerScript identifier and *procedure_name* is the name of a stored procedure in the database.

The parameter references can take the form of any valid parameter string the database accepts. PowerBuilder inspects the parameter list format only for variable substitution.

You must use the reserved word OUTPUT or OUT to indicate an output parameter if you want to get the output parameter value. If the stored procedure has a return value and you want to get it, use the syntax "RC=procedure_name". If the procedure has one or more result sets, only after all the result set has been retrieved can you get the output parameter or return value. The USING clause is required only if you are using a transaction object other than the default transaction object (SQLCA).

Example 1

Assume a stored procedure named proc1 is defined on the server. To declare proc1 for processing within PowerBuilder, enter:

```
DECLARE emp_proc PROCEDURE FOR proc1;
```

The procedure declaration is a nonexecutable statement, just like a cursor declaration. However, where cursors have an OPEN statement, procedures have an EXECUTE statement.

When an EXECUTE statement executes, the procedure is invoked. The EXECUTE refers to the logical procedure name, in this example emp_proc:

```
EXECUTE emp_proc;
```

Example 2

This example declares a stored procedure with two input and one output parameters:

```
DECLARE sp_duration PROCEDURE FOR pr_date_diff_prd_ken
   @var_date_1 = :ad_start,
   @var_date_2 = :ad_end,
   @rtn_diff_prd = :ls_duration OUTPUT;
```

If the stored procedure contains result sets, you must fetch the result sets first. If the stored procedure has a return value and you want to obtain it, use the format RC=*procedure_name*:

```
DECLARE sp_duration PROCEDURE FOR&
      RC=pr_date_diff_prd_ken
   @var_date_1 = :ad_start,
   @var_date_2 = :ad_end,
   @rtn_diff_prd = :ls_duration OUTPUT;
```

See also

OLE DB EXECUTE

OLE DB FETCH

To access rows returned in a result set, use the FETCH statement the same way you use it for cursors. The FETCH statement can be executed after any successful EXECUTE statement for a procedure that returns a result set.

Example

```
FETCH emp_proc INTO :emp_name_var;
```

Using FETCH after EXECUTE
Following an EXECUTE statement for a procedure, you can use the FETCH statement only to access values produced by the SELECT statement in the database stored procedure.

Since PowerBuilder cannot determine at compile time what result set will be returned when a database stored procedure executes, you must code FETCH statements so that the stored procedure exactly matches the format of the result set during execution. Assume you coded the second FETCH statement in the example above as:

```
FETCH emp_proc INTO :var1, :var2, :var3;
```

The statement compiles without errors. However, you will get an execution error indicating that the number of columns in the FETCH statement does not match the number of columns in the result set.

See also

OLE DB EXECUTE
OLE DB FETCH NEXT

OLE DB CLOSE

If a database stored procedure returns a result set, you must close the stored procedure when processing is complete. The procedure remains open until you close it, execute a COMMIT or ROLLBACK, or end the database connection.

Do you have to retrieve all the rows?
You do not have to retrieve all rows in a result set to close a request or procedure.

Example

Closing a procedure looks the same as closing a cursor:

```
CLOSE emp_proc;
```

OLE DB EXECUTE

Database stored procedures that perform only updates and do not return a result set are handled in much the same way as procedures that return a result set. The only difference is that after the EXECUTE *procedure_name* statement executes, no result set is pending, so a CLOSE statement is not required.

Using the SQLCode property

If a specific procedure can never return a result set, only the EXECUTE statement is required. If a procedure may or may not return a result set, you can test the SQLCode property of the referenced transaction object for +100 (the code for NOT FOUND) after the EXECUTE.

The possible values for SQLCode after an EXECUTE are:

Return code	Means
0	The EXECUTE was successful and a result set is pending. Regardless of the number of FETCH statements executed, the procedure must be explicitly closed with a CLOSE statement. This code is returned even if the result set is empty.
+100	Fetched row not found.
-1	The EXECUTE was not successful and no result set was returned.

Example 1

This example illustrates how to execute a stored procedure that does not return a result set. It assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed.

```
// good_employee is a database stored procedure.
// Declare the procedure.
DECLARE good_emp_proc PROCEDURE
    FOR good_employee;

// Execute it.
EXECUTE good_emp_proc;

// Test return code. Allow for +100 since you
// do not expect a result set.
if SQLCA.sqlcode = -1 then

// Issue an error message since it failed.
    MessageBox("Stored Procedure Error!", &
    SQLCA.sqlerrtext)
end if
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

This example illustrates how to pass parameters to a database stored procedure. It assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed. Emp_id_var was set to 691 elsewhere.

```
// get_employee is a database stored procedure.
// Declare the procedure.
DECLARE get_emp_proc PROCEDURE FOR
    get_employee @emp_id_parm = :emp_id_var;

// Declare a destination variable for emp_name.
```

```
string emp_name_var

// Execute the stored procedure using the
// current value for emp_id_var.
EXECUTE get_emp_proc;

// Test return code to see if it worked.
if SQLCA.sqlcode = 0 then

// Since we got a row, fetch it and display it.
   FETCH get_emp_proc INTO :emp_name_var;

// Display the employee name.
   MessageBox("Got my employee!",emp_name_var)

// You are all done, so close the procedure.
   CLOSE Get_emp_proc;
end if
```

OLE DB Using database stored procedures in DataWindow objects

You can use database stored procedures as a data source for DataWindow objects. The following rules apply:

- **Result set definition** You must define what the result set looks like in the DataWindow painter. PowerBuilder cannot determine this information from the stored procedure definition in the database.
- **Stored procedure arguments** The DataWindow painter provides the arguments for stored procedures only if the driver you are using to connect gives PowerBuilder the required information. If the arguments for the database stored procedure are not provided, you must define them.
- **DataWindow updates** Updates are not allowed for stored procedures in a DataWindow object. Only retrieval is allowed.

CHAPTER 19 Using Embedded SQL with ADO.NET

When you use the ADO.NET interface to connect to a database, you can use embedded SQL in your scripts.

ADO.NET DECLARE and EXECUTE

PowerBuilder requires a declarative statement to identify the database stored procedure that is being used and a logical name that can be referenced in subsequent SQL statements.

The general syntax for declaring a procedure is:

```
DECLARE logical_procedure_name PROCEDURE FOR
    procedure_name
    @Param1 = value1, @Param2 = value2 ,
    @Param3 = value3 OUTPUT,
    {USING transaction_object} ;
```

where *logical_procedure_name* can be any valid PowerScript data identifier and *procedure_name* is the name of the stored procedure in the database.

The parameter references can take the form of any valid parameter string that ADO.NET accepts. PowerBuilder does not inspect the parameter list format except for purposes of variable substitution. You must use the reserved word OUTPUT to indicate an output parameter. The USING clause is required only if you are using a transaction object other than the default transaction object (SQLCA).

Example 1

Assume a stored procedure proc1 is defined as:

```
CREATE PROCEDURE proc1 AS
    SELECT emp_name FROM employee
```

To declare that procedure for processing within PowerBuilder, enter:

```
DECLARE emp_proc PROCEDURE FOR proc1;
```

Note that this declaration is a nonexecutable statement, just like a cursor declaration. Where cursors have an OPEN statement, procedures have an EXECUTE statement.

When an EXECUTE statement executes, the procedure is invoked. The EXECUTE refers to the logical procedure name:

```
EXECUTE emp_proc;
```

Example 2

To declare a procedure with input and output parameters, enter:

```
DECLARE sp_duration PROCEDURE FOR pr_date_diff_prd_ken
    @var_date_1 = :ad_start,
    @var_date_2 = :ad_end,
    @rtn_diff_prd = :ls_duration OUTPUT;
```

CHAPTER 20

Using Embedded SQL with SAP Adaptive Server Enterprise

About this chapter

When you create scripts for a PowerBuilder application, you can use embedded SQL statements in the script to perform operations on the database. The features supported when you use embedded SQL depend on the DBMS to which your application connects.

Overview

When you use the SAP Adaptive Server Enterprise interface, you can use embedded SQL in your scripts. You can embed the following types of SQL statements in scripts and user-defined functions:

- Transaction management statements
- Non-cursor statements
- Cursor statements
- Database stored procedures

Client Library API

The SAP Adaptive Server Enterprise database interface uses the Client Library (CT-Lib) application programming interface (API) to interact with the database.

When you use embedded SQL, PowerBuilder makes the required calls to the API. Therefore, you do not need to know anything about CT-Lib to use embedded SQL in PowerBuilder.

See also

Chapter 8, Using Adaptive Server Enterprise
SAP Adaptive Server Enterprise SQL functions
SAP Adaptive Server Enterprise Transaction management statements
SAP Adaptive Server Enterprise Non-cursor statements
SAP Adaptive Server Enterprise Cursor statements
SAP Adaptive Server Enterprise Database stored procedures
SAP Adaptive Server Enterprise Name qualification

SAP Adaptive Server Enterprise Name qualification

Since PowerBuilder does not inspect all SQL statement syntax, you can qualify Adaptive Server Enterprise catalog entities as necessary.

For example, the following qualifications are all acceptable:

- emp_name
- employee.emp_name
- dbo.employee.emp_name
- emp_db.dbo.employee.emp_name

SAP Adaptive Server Enterprise SQL functions

You can use any function that Adaptive Server Enterprise supports (such as aggregate or mathematical functions) in SQL statements.

This example shows how to use the Adaptive Server Enterprise function UPPER in a SELECT statement:

```
SELECT UPPER(emp_name)
    INTO :emp_name_var
    FROM employee;
```

Calling Client Library functions

While PowerBuilder provides access to a large percentage of the features within Adaptive Server Enterprise, in some cases you may decide that you need to call one or more Client Library (CT-Lib) functions directly for a particular application. PowerBuilder provides access to any Windows DLL by using external function declarations.

CT-Lib calls require a pointer to one of the following structures as their first parameter:

- CS_CONNECTION
- CS_CONTEXT
- CS_COMMAND

You can obtain the current CS_CONNECTION pointer by using the PowerScript DBHandle function.

Using DBHandle to obtain the CS_CONNECTION pointer

DBHandle takes a transaction object as a parameter and returns a long variable, which is the CS_CONNECTION pointer that PowerBuilder uses internally to communicate with the database. You can pass this value as one of the parameters to your external function.

This example shows how to use DBHandle. Assume a successful connection has occurred using the default transaction object (SQLCA):

```
// Define a variable to hold our DB handle.
long     SQLServerHandle

// Go get the handle.
SQLServerHandle = SQLCA.DBHandle( )

// Now that you have the CS_CONNECTION pointer,
// call the DLL function.
MyDLLFunction( SQLServerHandle, parm1, parm2, ... )
```

In your DLL, cast the incoming long value into a pointer to a CS_CONNECTION structure:

```
MyDLLFunction( long 1SQLServerHandle,
        parm1_type parm1,
        parm2_type Parm2, ... )
{
CS_CONNECTION * pConnect;
pConnect = (CS_CONNECTION *)  1SQLServerHandle;
// CT-LIB functions can be called using pConnect.
}
```

Obtaining the CS_CONTEXT pointer

Within your external function, you can obtain the CS_CONTEXT pointer with the following function call:

```
CS_RETCODE        RC;
CS_CONNECTION     * PConnect;
CS_INT            outlen;
CS_CONTEXT         * pContext;
rc = ct_con_props (pConnect,CS_GET,CS_PARENT_HANDLE,
                  (CS_VOID *) &pContext, CS_UNUSED,
                  &outlen);
```

Allocating a new command pointer

Likewise, you can allocate a new command pointer with the following code:

```
CS_COMMAND     * pCommand;
rc = ct_cmd_alloc(pConnect, &pCommand);
```

SAP Adaptive Server Enterprise Transaction management statements

You use the following transaction management statements with transaction objects to manage connections and transactions for Adaptive Server Enterprise databases:

- CONNECT
- COMMIT
- DISCONNECT
- ROLLBACK

Transaction management statements in triggers

You should not use transaction statements in triggers. A trigger is a special kind of stored procedure that takes effect when you issue a statement such as INSERT, DELETE, or UPDATE on a specified table or column. Triggers can be used to enforce referential integrity.

For example, assume that a certain condition within a trigger is not met and you want to execute a ROLLBACK. Instead of coding the ROLLBACK directly in the trigger, you should use RAISERROR and test for that particular return code in the DBMS-specific return code (SQLDBCode) property within the referenced transaction object.

See also

SAP Adaptive Server Enterprise Using CONNECT, COMMIT, DISCONNECT, and ROLLBACK

SAP Adaptive Server Enterprise Using CONNECT, COMMIT, DISCONNECT, and ROLLBACK

The following table lists each transaction management statement and describes how it works when you use the SAP Adaptive Server Enterprise interface to connect to a database:

Statement	Description
CONNECT	Establishes the database connection. After you assign values to the required properties of the transaction object, you can execute a CONNECT. After the CONNECT completes successfully, PowerBuilder automatically starts a transaction. This is the start of a logical unit of work. If AutoCommit is true, PowerBuilder does not start a transaction.

Statement	Description
COMMIT	COMMIT terminates the logical unit of work, guarantees that all changes made to the database since the beginning of the current unit of work become permanent, and starts a new logical unit of work. If AutoCommit is false (the default), a COMMIT TRANSACTION executes, then a BEGIN TRANSACTION executes to start a new logical unit of work. If AutoCommit is true, the COMMIT is issued but has no effect because all previous database changes were already automatically committed.
DISCONNECT	Terminates a successful connection. DISCONNECT automatically executes a COMMIT to guarantee that all changes made to the database since the beginning of the current unit of work are committed. If AutoCommit is false, a COMMIT TRANSACTION executes automatically to guarantee that all changes made to the database since the beginning of the current logical unit of work are committed.
ROLLBACK	ROLLBACK terminates a logical unit of work, undoes all changes made to the database since the beginning of the logical unit of work, and starts a new logical unit of work. If AutoCommit is false, a ROLLBACK TRANSACTION executes, then a BEGIN TRANSACTION executes to start a new logical unit of work. If AutoCommit is true, a ROLLBACK TRAN executes but has no effect because all previous database changes were already committed.

See also

SAP Adaptive Server Enterprise Performance and locking
SAP Adaptive Server Enterprise Using AutoCommit

SAP Adaptive Server Enterprise Using AutoCommit

The setting of the AutoCommit property of the transaction object determines whether PowerBuilder issues SQL statements inside or outside the scope of a transaction. When AutoCommit is set to false or 0 (the default), SQL statements are issued inside the scope of a transaction. When you set AutoCommit to true or 1, SQL statements are issued outside the scope of a transaction.

Adaptive Server Enterprise requires you to execute Data Definition Language (DDL) statements outside the scope of a transaction unless you set the database option "ddl in tran" to true. If you execute a database stored procedure that contains DDL statements within the scope of a transaction, an error message is returned and the DDL statements are rejected. When you use the transaction object to execute a database stored procedure that creates a temporary table, you do not want to associate the connection with a transaction.

To execute Adaptive Server Enterprise stored procedures containing DDL statements, you must either set "ddl in tran" to true, or set AutoCommit to true so PowerBuilder issues the statements outside the scope of a transaction. However, if AutoCommit is set to true, you cannot issue a ROLLBACK. Therefore, you should set AutoCommit back to false (the default) immediately after completing the DDL operation.

When you change the value of AutoCommit from false to true, PowerBuilder issues a COMMIT statement by default.

See also

SAP Adaptive Server Enterprise Performance and locking
SAP Adaptive Server Enterprise Using CONNECT, COMMIT, DISCONNECT, and ROLLBACK

SAP Adaptive Server Enterprise Performance and locking

An important consideration when designing a database application is deciding when CONNECT and COMMIT statements should occur to maximize performance and limit locking and resource use. A CONNECT takes a certain amount of time and can tie up resources during the life of the connection. If this time is significant, then limiting the number of CONNECT statements is desirable.

In addition, after a connection is established, SQL statements can cause locks to be placed on database entities. The more locks at a given moment in time, the more likely it is that the locks will hold up another transaction.

Rules

No set of rules for designing a database application is totally comprehensive. However, when you design a PowerBuilder application, you should do the following:

- **Long-running connections** Determine whether you can afford to have long-running connections. If not, your application should connect to the database only when absolutely necessary. After all the work for that connection is complete, the transaction should be disconnected.

  If long-running connections are acceptable, then COMMITs should be issued as often as possible to guarantee that all changes do in fact occur. More importantly, COMMITs should be issued to release any locks that may have been placed on database entities as a result of the statements executed using the connection.

- **SetTrans or SetTransObject function** Determine whether you want to use default DataWindow transaction processing (the SetTrans function) or control the transaction in a script (the SetTransObject function).

  If you cannot afford to have long-running connections and therefore have many short-lived transactions, use the default DataWindow transaction processing. If you want to keep connections open and issue periodic COMMITs, use the SetTransObject function and control the transaction yourself.

Isolation feature

SAP Adaptive Server Enterprise databases use the isolation feature to support assorted database lock options. In PowerBuilder, you can use the Lock property of the transaction object to set the isolation level when you connect to the database.

The following example shows how to set the Lock property to Read uncommitted:

Example 1

This script uses embedded SQL to connect to a database and insert a row in the ORDER_HEADER table and a row in the ORDER_ITEM table. Depending on the success of the statements in the script, the script executes either a COMMIT or a ROLLBACK.

```
// Set the SQLCA connection properties.
SQLCA.DBMS = "SYC"
SQLCA.servername = "SERVER24"
SQLCA.database = "ORDERS"
SQLCA.logid = "JPL"
SQLCA.logpass = "TREESTUMP"

// Connect to the database. AutoCommit is set to
// False by default.
CONNECT USING SQLCA;

// Insert a row into the ORDER_HEADER table.
// A ROLLBACK is required only if the first row
// was inserted successfully.
```

```
INSERT INTO ORDER_HEADER (ORDER_ID,CUSTOMER_ID)
    VALUES ( 7891, 129 );

// Test return code for ORDER_HEADER insertion.
if SQLCA.sqlcode = 0 then

// Since the ORDER_HEADER is inserted,
// try to insert ORDER_ITEM.
INSERT INTO ORDER_ITEM(ORDER_ID, ITEM_NBR,
    PART_NBR, QTY)
       VALUES ( 7891, 1, '991PLS', 456 );

// Test return code for ORDER_ITEM insertion.
    if SQLCA.sqlcode = -1 then

// If insert failed, roll back insertion of
// ORDER_HEADER.
    ROLLBACK USING SQLCA;
    end if
end if

// Commit changes and disconnect from the database.
DISCONNECT USING SQLCA;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

This example uses the scripts for the Open and Close events in a window and the Clicked event in a CommandButton to illustrate how you can manage transactions in a DataWindow control. Assume that the window contains a DataWindow control dw_1 and that the user enters data in dw_1 and then clicks the Cb_Update button to send the data to the database.

Since this script uses SetTransObject to connect to the database, the programmer is responsible for managing the transaction.

The window OPEN event script:

```
// Set the transaction object properties
// and connect to the database.
// Set the SQLCA connection properties.
SQLCA.DBMS = "SYC"
SQLCA.servername = "SERVER24"
SQLCA.database = "ORDERS"
SQLCA.logid = "JPL"
SQLCA.logpass = "TREESTUMP"
```

```
// Connect to the database.
CONNECT USING SQLCA;

// Tell the DataWindow which transaction object
// to use.
SetTransObject( dw_1, SQLCA )
```

The CommandButton CLICKED event script:

```
// Declare ReturnValue an integer.
integer ReturnValue
ReturnValue = Update( dw_1 )

// Test to see if updates were successful.
if ReturnValue = -1 then

// Updates were not successful. Since we used
// SetTransObject, roll back any changes made
// to the database.
    ROLLBACK USING SQLCA;
else

// Updates were successful. Since we used
// SetTransObject, commit any changes made
// to the database.
    COMMIT USING SQLCA;
end if
```

The window CLOSE event script:

```
// Disconnect from the database.
DISCONNECT USING SQLCA;
```

See also

SAP Adaptive Server Enterprise Using CONNECT, COMMIT, DISCONNECT, and ROLLBACK

SAP Adaptive Server Enterprise Non-cursor statements

The statements that do not involve cursors or procedures are:

- DELETE (SAP Adaptive Server Enterprise DELETE, INSERT, and UPDATE)
- INSERT (SAP Adaptive Server Enterprise DELETE, INSERT, and UPDATE)

- SELECT (singleton) (SAP Adaptive Server Enterprise SELECT)
- UPDATE (SAP Adaptive Server Enterprise DELETE, INSERT, and UPDATE)

SAP Adaptive Server Enterprise DELETE, INSERT, and UPDATE

Internally, PowerBuilder processes DELETE, INSERT, and UPDATE statements the same way. PowerBuilder inspects them for any PowerScript data variable references and replaces all such references with a constant that conforms to Adaptive Server Enterprise rules for the data type.

Example

Assume you enter the following statement:

```
DELETE FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var is a PowerScript data variable with the data type of integer that has been defined within the scope of the script that contains the DELETE statement. Before the DELETE statement is executed, emp_id_var is assigned a value (say 691) so that when the DELETE statement executes, the database receives the following statement:

```
DELETE FROM employee WHERE emp_id = 691;
```

When is this substitution technique used?

This variable substitution technique is used for all PowerScript variable types. When you use embedded SQL, precede all PowerScript variables with a colon (:).

See also

SAP Adaptive Server Enterprise SELECT

SAP Adaptive Server Enterprise SELECT

The SELECT statement contains input variables and output variables.

- Input variables are passed to the database as part of the execution and the substitution as described above for DELETE, INSERT, and UPDATE.
- Output variables are used to return values based on the result of the SELECT statement.

Example 1

Assume you enter the following statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var, emp_salary_var, and emp_name_var are variables defined within the scope of the script that contains the SELECT statement, and emp_id_var is processed as described in the DELETE example above.

Both emp_name_var and emp_salary_var are output variables that will be used to return values from the database. The data types of emp_name_var and emp_salary_var should be the PowerScript data types that best match the Adaptive Server Enterprise data type. When the data types do not match perfectly, PowerBuilder converts them.

How big should numeric output variables be?
For numeric data, the output variable must be large enough to hold any value that may come from the database.

Assume the value for emp_id_var is 691 as in the previous example. When the SELECT statement executes, the database receives the following statement:

```
SELECT emp_name, emp_salary
    FROM employee WHERE emp_id = 691;
```

If the statement executes with no errors, data locations for the result fields are bound internally. The data returned into these locations is then converted as necessary and the appropriate PowerScript data variables are set to those values.

Example 2

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. It also assumes the data type of the emp_id column in the employee table is CHARACTER[10].

The user enters an employee ID into the single line edit field sle_Emp and clicks the button Cb_Delete to delete the employee.

The script for the Clicked event in the CommandButton Cb_Delete is:

```
// Make sure we have a value.
if sle_Emp.text <> "" then

// Since we have a value, try to delete it.
    DELETE FROM employee
    WHERE emp_id = :sle_Emp.text;
```

```
// Test to see if the DELETE worked.
   if SQLCA.sqlcode = 0 then

// It seems to have worked, let user know.
      MessageBox( "Delete",&
      "The delete has been successfully "&
      +" processed!")
      COMMIT;
   else

//It didn't work.
      MessageBox( "Error", &
      "The delete failed. Employee ID is not "&
      +"valid.")
      ROLLBACK;
   end if
else

// No input value. Prompt user.
      MessageBox( "Error",&
      "An employee ID is required for "&
      +"delete!" )
   end if
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 3

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. The user wants to extract rows from the employee table and insert them into the table named extract_employees. The extraction occurs when the user clicks the button Cb_Extract. The boolean variable YoungWorkers is set to TRUE or FALSE elsewhere in the application.

The script for the Clicked event for the CommandButton Cb_Extract is:

```
integer    EmployeeAgeLowerLimit
integer    EmployeeAgeUpperLimit

// Do they have young workers?
if ( YoungWorkers = TRUE ) then

// Yes - set the age limit in the YOUNG range.
// Assume no employee is under legal working age.
   EmployeeAgeLowerLimit = 16
```

```
// Pick an upper limit.
   EmployeeAgeUpperLimit = 42
else

// No - set the age limit in the OLDER range.
   EmployeeAgeLowerLimit = 43

// Pick an upper limit that includes all
// employees.
   EmployeeAgeUpperLimit = 200
end if

INSERT INTO extract_employee(emp_id,emp_name)
   SELECT emp_id, emp_name FROM employee
      WHERE emp_age >= :EmployeeAgeLowerLimit
      AND emp_age <= :EmployeeAgeUpperLimit;

// If there are no errors, commit the changes.
if SQLCA.sqlcode = 0 then
   COMMIT;
else

// If there are errors, roll back the changes and
// tell the user.
   ROLLBACK;
   MessageBox( "Insert Failed", SQLCA.sqlerrtext)
end if
```

See also

SAP Adaptive Server Enterprise DELETE, INSERT, and UPDATE

SAP Adaptive Server Enterprise Cursor statements

In embedded SQL, statements that retrieve data and statements that update data can both involve cursors.

Retrieval statements

The retrieval statements that involve cursors are:

- DECLARE *cursor_name* CURSOR FOR ...
- OPEN *cursor_name*
- FETCH *cursor_name* INTO ...
- CLOSE *cursor_name*

Update statements

The update statements that involve cursors are:

- UPDATE ... WHERE CURRENT OF *cursor_name*

- DELETE ... WHERE CURRENT OF *cursor_name*

Setting CursorUpdate to use updatable cursors

To use the `UPDATE ... WHERE CURRENT OF` or `DELETE ... WHERE CURRENT OF` statements, you must set the CursorUpdate DBParm parameter to 1 before declaring the cursor. (By default, CursorUpdate is set to 0.)

For example:

```
SQLCA.DBParm = "CursorUpdate = 1"
```

You can set the CursorUpdate parameter at any time before or after connecting to the database. You can also change its setting at any time.

See also

SAP Adaptive Server Enterprise Retrieval Using Cursors
SAP Adaptive Server Enterprise Closing the Cursor

SAP Adaptive Server Enterprise Retrieval Using Cursors

Retrieval using cursors is conceptually similar to retrieval in the singleton SELECT. The main difference is that since there can be multiple rows in a result set, you control when the next row is fetched into the PowerScript data variables.

If you expect only a single row to exist in the employee table with the specified emp_id, use the singleton SELECT. In a singleton SELECT, you specify the SELECT statement and destination variables in one concise SQL statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

However, when a SELECT may return multiple rows, you must:

1. Declare a cursor.
2. Open it (which conceptually executes the SELECT).
3. Fetch rows as needed.
4. Close the cursor.

Declaring and opening a cursor

Declaring a cursor is tightly coupled with the OPEN statement. The DECLARE specifies the SELECT statement to be executed, and the OPEN actually executes it.

Declaring a cursor is similar to declaring a variable; a cursor is a nonexecutable statement just like a variable declaration. The first step in declaring a cursor is to define how the result set looks. To do this, you need a SELECT statement. Since you must refer to the result set in subsequent SQL statements, you must associate the result set with a logical name.

Multiple cursors
The CT-Lib API lets you declare and open multiple cursors without having to open additional database connections.

Example

Assume the SingleLineEdit sle_1 contains the state code for the retrieval:

```
// Declare cursor emp_curs for employee table retrieval.
DECLARE emp_curs CURSOR FOR
    SELECT emp_id, emp_name FROM EMPLOYEE
    WHERE emp_state = :sle_1.text;

// Declare local variables for retrieval.
string emp_id_var
string emp_name_var

// Execute the SELECT statement with
// the current value of sle_1.text.
OPEN emp_curs;

// At this point, if there are no errors,
// the cursor is available for further
// processing.
```

Fetching rows

In the singleton SELECT, you specify variables to hold the values for the columns within the selected row. The FETCH statement syntax is similar to the syntax of the singleton SELECT. Values are returned INTO a specified list of variables.

This example continues the previous example by retrieving some data:

```
// Go get the first row from the result set.
FETCH emp_curs INTO :emp_id_var, :emp_name_var;
```

If at least one row can be retrieved, this FETCH places the values of the emp_id and emp_name columns from the first row in the result set into the PowerScript data variables emp_id_var and emp_name_var. Executing another FETCH statement will place the variables from the next row into specified variables.

FETCH statements typically occur in a loop that processes several rows from a result set (one row at a time): fetch the row, process the variables, and then fetch the next row.

What happens when the result set is exhausted?
FETCH returns +100 (not found) in the SQLCode property within the referenced transaction object. This is an informational return code; -1 in SQLCode indicates an error.

Example

This cursor example illustrates how you can loop through a result set. Assume the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed.

The statements retrieve rows from the employee table and then display a message box with the employee name in each row that is found.

```
// Declare the emp_curs.
DECLARE emp_curs CURSOR FOR
    SELECT emp_name FROM EMPLOYEE
    WHERE emp_state = :sle_1.text;

// Declare a destination variable for employee
// names.
string    emp_name_var

// Execute the SELECT statement with the
// current value of sle_1.text.
OPEN emp_curs;

// Fetch the first row from the result set.
FETCH emp_curs INTO :emp_name_var;

// Loop through result set until exhausted.
DO WHILE SQLCA.sqlcode = 0

// Pop up a message box with the employee name.
    MessageBox("Found an employee!",emp_name_var)

// Fetch the next row from the result set.
    FETCH emp_curs INTO :emp_name_var;
LOOP
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

SAP Adaptive Server Enterprise Closing the Cursor

The CLOSE statement terminates processing for the specified cursor. CLOSE releases resources associated with the cursor, and subsequent references to that cursor are allowed only if another OPEN is executed. Although you can have multiple cursors open at the same time, you should close the cursors as soon as possible for efficiency reasons.

Unlike the DB-Library interface to SQL Server, the CT-Library interface lets you issue other commands while a cursor is open.

Example

In this example, the additional request for the employee name (shown in bold) is issued while the cursor is open. Under the DB-Library interface, this request would have failed and returned a Results Pending message. Under the CT-Library interface, it succeeds.

```
string    dname
long      depthead
string    fname
string    lname

SQLCA.dbms = "SYC"
SQLCA.database = "mzctest"
SQLCA.logid = "mikec"
SQLCA.logpass = "mikecx"
SQLCA.servername = "SYB1001"
SQLCA.autocommit = "false"

CONNECT USING SQLCA;
if SQLCA.sqlcode <> 0 then
   MessageBox("Connect Error",SQLCA.sqlerrtext)
end if

DECLARE dept_curs CURSOR FOR SELECT dept_name,
   dept_head_id FROM department;
OPEN dept_curs;
if SQLCA.sqlcode < 0 then
   MessageBox("Open Cursor",SQLCA.sqlerrtext)
end if

DO WHILE SQLCA.sqlcode = 0
   FETCH dept_curs INTO :dname, :depthead;
   if SQLCA.sqlcode < 0 then
      MessageBox("Fetch Error",SQLCA.sqlerrtext)
   elseif SQLCA.sqlcode = 0 then
      SELECT emp_fname, emp_lname INTO
         :fname,:lname FROM employee
         WHERE emp_id = :depthead;
```

```
            if SQLCA.sqlcode <> 0 then
               MessageBox("Singleton Select",  &
                  SQLCA.sqlerrtext)
            end if
         end if
      LOOP

      CLOSE dept_curs;
      if SQLCA.sqlcode <> 0 then
         MessageBox("Close Cursor", SQLCA.sqlerrtext)
      end if
```

See also

SAP Adaptive Server Enterprise SELECT

SAP Adaptive Server Enterprise Database stored procedures

One of the most significant features of SAP Adaptive Server Enterprise is database stored procedures. You can use database stored procedures for:

- Retrieval only
- Update only
- Update and retrieval

PowerBuilder supports all these uses in embedded SQL.

Using AutoCommit with database stored procedures

The setting of the AutoCommit property of the transaction object determines whether PowerBuilder issues SQL statements inside or outside the scope of a transaction. When AutoCommit is set to false or 0 (the default), SQL statements are issued inside the scope of a transaction. When you set AutoCommit to true or 1, SQL statements are issued outside the scope of a transaction.

Adaptive Server Enterprise requires you to execute Data Definition Language (DDL) statements outside the scope of a transaction unless you set the database option "ddl in tran" to true. If you execute a database stored procedure that contains DDL statements within the scope of a transaction, an error message is returned and the DDL statements are rejected. When you use the transaction object to execute a database stored procedure that creates a temporary table, you do not want to associate the connection with a transaction.

To execute Adaptive Server Enterprise stored procedures containing DDL statements, you must either set "ddl in tran" to true, or set AutoCommit to true so PowerBuilder issues the statements outside the scope of a transaction. However, if AutoCommit is set to true, you cannot issue a ROLLBACK. Therefore, you should set AutoCommit back to false (the default) immediately after completing the DDL operation.

When you change the value of AutoCommit from false to true, PowerBuilder issues a COMMIT statement by default.

Using transaction statements in database stored procedures

Transaction statements in database stored procedures are not honored when the stored procedure is executing within the scope of a transaction. For example, a ROLLBACK statement will not be honored if the following are all true:

- The AutoCommit property is FALSE (process transactions normally) when the transaction is connected.
- The database stored procedure executes using a transaction.
- The procedure contains a ROLLBACK statement.

You should use alternative means to execute the ROLLBACK. For example, you can use return values as described in the information about triggers in Transaction management statements (SAP Adaptive Server Enterprise Transaction management statements).

See also

SAP Adaptive Server Enterprise Retrieval
SAP Adaptive Server Enterprise Temporary tables
SAP Adaptive Server Enterprise Update
SAP Adaptive Server Enterprise Return values and output parameters
SAP Adaptive Server Enterprise System stored procedures
SAP Adaptive Server Enterprise Using database stored procedures in DataWindow objects

SAP Adaptive Server Enterprise Retrieval

PowerBuilder uses a construct that is very similar to cursors to support retrieval using database stored procedures. In the PowerBuilder-supported embedded SQL, there are four commands that involve database stored procedures:

- DECLARE *procedure_name* PROCEDURE FOR ...
- EXECUTE *procedure_name*
- FETCH *procedure_name* INTO ...

- CLOSE *procedure_name*

See also

SAP Adaptive Server Enterprise DECLARE and EXECUTE
SAP Adaptive Server Enterprise FETCH
SAP Adaptive Server Enterprise CLOSE

SAP Adaptive Server Enterprise DECLARE and EXECUTE

PowerBuilder requires a declarative statement to identify the database stored procedure that is being used and a logical name that can be referenced in subsequent SQL statements.

The general syntax for declaring a procedure is:

```
DECLARE logical_procedure_name PROCEDURE FOR
    {@rv = } SQL_Server_procedure_name
    @Param1 = value1, @Param2 = value2 , ...
    {USING transaction_object} ;
```

where *logical_procedure_name* can be any valid PowerScript data identifier, *SQL_Server_procedure_name* is the name of the stored procedure in the database, and @rv is an optional return value.

The parameter references can take the form of any valid parameter string that Adaptive Server Enterprise accepts. PowerBuilder does not inspect the parameter list format except for purposes of variable substitution. The USING clause is required only if you are using a transaction object other than the default transaction object (SQLCA).

Example

Assume a stored procedure proc1 is defined as:

```
CREATE PROCEDURE proc1 AS
    SELECT emp_name FROM employee
```

To declare that procedure for processing within PowerBuilder, enter:

```
DECLARE emp_proc PROCEDURE FOR proc1;
```

Note that this declaration is a nonexecutable statement, just like a cursor declaration. Where cursors have an OPEN statement, procedures have an EXECUTE statement.

When an EXECUTE statement executes, the procedure is invoked. The EXECUTE refers to the logical procedure name:

```
EXECUTE emp_proc;
```

SAP Adaptive Server Enterprise FETCH

To access rows returned in a result set, you use the FETCH statement the same way you use it for cursors. The FETCH statement can be executed after any EXECUTE statement that refers to a procedure that returns a result set.

For example:

```
FETCH emp_proc INTO :emp_name_var;
```

Note You can use this FETCH statement only to access values produced with a SELECT statement in a database stored procedure. You cannot use the FETCH statement to access computed rows.

Example 1

Database stored procedures can return multiple result sets. Assume you define a database stored procedure proc2 as:

```
CREATE PROCEDURE proc2 AS
    SELECT emp_name FROM employee
    SELECT part_name FROM parts
```

PowerBuilder provides access to both result sets:

```
// Declare the procedure.
DECLARE emp_proc2 PROCEDURE FOR proc2;

// Declare some variables to hold results.
string    emp_name_var
string    part_name_var

// Execute the stored procedure.
EXECUTE emp_proc2;

// Loop through all rows in the first result
// set.
DO WHILE SQLCA.sqlcode = 0

// Fetch the next row from the first result set.
    FETCH emp_proc2 INTO :emp_name_var;
LOOP

// At this point we have exhausted the first
// result set. After this occurs,
// PowerBuilder notes that there is another
// result set and internally shifts result sets.
// The next FETCH executed will retrieve the
// first row from the second result set.
// Fetch the first row from the second result
// set.
if SQLCA.sqlcode = 100 then
   FETCH emp_proc2 INTO :part_name_var;
end if

// Loop through all rows in the second result
// set.
DO WHILE SQLCA.sqlcode = 0

// Fetch the next row from the second result
// set.
   FETCH emp_proc2 INTO :part_name_var;
LOOP

// Close the procedure.
CLOSE emp_proc2;
```

The result sets that will be returned when a database stored procedure executes cannot be determined at compile time. Therefore, you must code FETCH statements that exactly match the format of a result set returned by the stored procedure when it executes.

Example 2

In the preceding example, if instead of coding the second fetch statement as:

```
FETCH emp_proc2 INTO :part_name_var;
```

you coded it as:

```
FETCH emp_proc2
   INTO :part_var1,:part_var2,:part_var3;
```

the statement would compile without errors. But an execution error would occur: the number of columns in the FETCH statement does not match the number of columns in the current result set. The second result set returns values from only one column.

SAP Adaptive Server Enterprise CLOSE

If a database stored procedure returns a result set, it must be closed when processing is complete. You do not have to retrieve all the rows in a result set to close a request or procedure.

Closing a procedure looks the same as closing a cursor:

```
CLOSE emp_proc;
```

If a procedure executes successfully and returns at least one result set and is not closed, a result set is pending and no SQL commands other than the FETCH can be executed. Procedures with result sets should be closed as soon as possible.

SAP Adaptive Server Enterprise Update

Database stored procedures that perform updates only and do not return result sets are handled in much the same way as procedures that return result sets. The only difference is that after the EXECUTE *procedure_name* statement is executed, no result sets are pending and no CLOSE statement is required.

Using the SQL Code property

If you know for sure that a particular procedure can never return result sets, then the EXECUTE statement is all that is needed. If there is a procedure that may or may not return a result set, you can test the SQLCode property of the referenced transaction object for +100 (the code for NOT FOUND) after the EXECUTE.

The following table shows all the possible values for SQLCode after an EXECUTE:

Return code	Means
0	The EXECUTE was successful and at least one result set is pending. Regardless of the number of FETCH statements executed, the procedure must be explicitly closed with a CLOSE statement. This code is returned even if the result set is empty.
+100	Fetched row not found.
-1	The EXECUTE was not successful and no result sets were returned. The procedure does not require a CLOSE. If a CLOSE is attempted against this procedure an error will be returned.

Example 1

Assume the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed. Also assume the description of the Adaptive Server Enterprise procedure good_employee is:

```
// Adaptive Server Enterprise good_employee
// stored procedure:
CREATE PROCEDURE good_employee AS
```

```
    UPDATE employee
    SET emp_salary=emp_salary * 1.1
    WHERE emp_status = 'EXC'
```

This example illustrates how to execute a stored procedure that does not return any result sets:

```
// Declare the procedure.
DECLARE good_emp_proc PROCEDURE
FOR good_employee;

// Execute it.
EXECUTE good_emp_proc;

// Test return code. Allow for +100 since you do
// not expect result sets.
if SQLCA.sqlcode = -1 then

// Issue error message since it failed.
    MessageBox("Stored Procedure Error!", &
    SQLCA.sqlerrtext)
end if
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

Assume the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed. Also assume the description of the Adaptive Server Enterprise procedure get_employee is:

```
// Adaptive Server Enterprise get_employee
// stored procedure:
    CREATE PROCEDURE get_employee @emp_id_parm
    int AS SELECT emp_name FROM employee
    WHERE emp_id = @emp_id_parm
```

This example illustrates how to pass parameters to a database stored procedure. *Emp_id_var* has been set elsewhere to 691:

```
// Declare the procedure.
DECLARE get_emp_proc PROCEDURE FOR
    get_employee @emp_id_parm = :emp_id_var;

// Declare a destination variable for emp_name.
string    emp_name_var

// Execute the stored procedure using the
// current value for emp_id_var.
EXECUTE get_emp_proc;

// Test return code to see if it worked.
if SQLCA.sqlcode = 0 then

// Since we got a row, fetch it and display it.
    FETCH get_emp_proc INTO :emp_name_var;

// Display the employee name.
    MessageBox("Got my employee!",emp_name_var)

// You are all done, so close the procedure.
    CLOSE Get_emp_proc;

end if
```

SAP Adaptive Server Enterprise Return values and output parameters

In addition to result sets, SAP Adaptive Server Enterprise stored procedures may return a long integer return value and output parameters of any data type. After all of the result sets have been returned, PowerScript requires you to issue one final FETCH procedure_name INTO . . . statement to obtain these values. The order in which these values are returned is:

```
return value, output parm1, output parm2, ...
```

Example 1

The following stored procedure contains one input parameter (@deptno) and returns a result set containing employee names and salaries for that department. It also returns two output parameters (@totsal and @avgsal), and a return value that is the count of employees in the department.

```
integer fetchcount = 0
long    lDeptno, rc
string  fname, lname
double  dSalary, dTotSal, dAvgSal

lDeptno = 100

DECLARE deptproc PROCEDURE FOR
    @rc = dbo.deptroster
    @deptno = :lDeptno,
    @totsal = 0 output,
    @avgsal = 0 output
USING SQLCA;
```

```
EXECUTE deptproc;
CHOOSE CASE SQLCA.sqlcode
CASE 0
   // Execute successful. There is at least one
   // result set. Loop to get the query result set
   // from the table SELECT.
   DO
      FETCH deptproc INTO :fname, :lname, :dSalary;
      CHOOSE CASE SQLCA.sqlcode
      CASE 0
         fetchcount++
      CASE 100
         MessageBox ("End of Result Set", &
            string (fetchcount) " rows fetched")
      CASE -1
         MessageBox ("Fetch Failed", &
            string (SQLCA.sqldbcode) " = " &
            SQLCA.sqlerrtext)
      END CHOOSE
   LOOP WHILE SQLCA.sqlcode = 0

   // Issue an extra FETCH to get the Return Value
   // and Output Parameters.
   FETCH deptproc INTO :rc, :dTotSal, :dAvgSal;
   CHOOSE CASE SQLCA.sqlcode
   CASE 0
      MessageBox ("Fetch Return Value and Output" &
         "Parms SUCCESSFUL", "Return Value is: " &
         string (rc) &
         "~r~nTotal Salary: " string (dTotSal) &
         "~r~nAverage Sal:  " string (dAvgSal))
   CASE 100
      MessageBox ("Return Value and Output Parms" &
         "NOT FOUND", "")
   CASE ELSE
      MessageBox ("Fetch Return Value and Output" &
         "Parms FAILED", "SQLDBCode is " &
         string (SQLCA.sqldbcode) " = " &
         SQLCA.sqlerrtext)
   END CHOOSE

   CLOSE deptproc;

CASE 100
   // Execute successful; no result set.
   // Do not try to close.
   MessageBox ("Execute Successful", "No result set")
```

```
CASE ELSE
    MessageBox ("Execute Failed", &
        string (SQLCA.sqldbcode) " = " &
        SQLCA.sqlerrtext)

END CHOOSE
```

SAP Adaptive Server Enterprise Temporary tables

Database stored procedures frequently contain temporary tables that are used as repositories when accumulating rows during processing within the procedure. Since Adaptive Server Enterprise requires you to execute Data Definition Language (DDL) statements outside the scope of a transaction unless you set the database option "ddl in tran" to true, PowerBuilder provides the boolean AutoCommit property in the transaction object to allow you to handle these cases.

The setting of AutoCommit determines whether PowerBuilder issues SQL statements inside or outside the scope of a transaction. When AutoCommit is set to false or 0 (the default), SQL statements are issued inside the scope of a transaction. When you set AutoCommit to true or 1, SQL statements are issued outside the scope of a transaction.

To execute Adaptive Server Enterprise stored procedures containing DDL statements, you must either set "ddl in tran" to true, or set AutoCommit to true so PowerBuilder issues the statements outside the scope of a transaction. However, if AutoCommit is set to true, you cannot issue a ROLLBACK. Therefore, you should set AutoCommit back to false (the default) immediately after completing the DDL operation.

When you change the value of AutoCommit from false to true, PowerBuilder issues a COMMIT statement.

SAP Adaptive Server Enterprise System stored procedures

You can access system database stored procedures the same way you access user-defined stored procedures. You can use the DECLARE statement against any procedure and can qualify procedure names if necessary.

SAP Adaptive Server Enterprise Using database stored procedures in DataWindow objects

Using stored procedures as DataWindow data sources

You can use a database stored procedures as a data source for DataWindow objects. The following rules apply:

- **Result set definition** You must define what the result set looks like. The DataWindow object cannot determine this information from the stored procedure definition in the database.
- **DataWindow updates** You cannot perform DataWindow updates through stored procedures (that is, you cannot update the database with changes made in the DataWindow object); only retrieval is allowed. (However, the DataWindow can have update characteristics set manually through the DataWindow painter.)
- **Result set processing** You can specify only one result set to be processed when you define the stored procedure result set in the DataWindow painter. However, the result set you select does not have to be the first result set.
- **Computed rows** Computed rows cannot be processed in a DataWindow.

Database stored procedures summary

When you use database stored procedures in a PowerBuilder application, keep the following points in mind:

- **Manipulating stored procedures** To manipulate database stored procedures, PowerBuilder provides SQL statements that are similar to cursor statements.
- **Retrieval and update** PowerBuilder supports retrieval, update, or a combination of retrieval and update in database stored procedures, including procedures that return no results sets and those that return one or more result sets.
- **Transactions and stored procedures with result sets** When a procedure executes successfully using a specific connection (transaction) and returns at least one result set, no other SQL commands can be executed using that connection until the procedure has been closed.
- **Transactions and stored procedures without result sets** When a procedure executes successfully using a specific transaction but does not return a result set, the procedure is no longer active. No result sets are pending, and therefore you should not execute a CLOSE statement.

CHAPTER 21

Using Embedded SQL with Informix

About this chapter

When you create scripts for a PowerBuilder application, you can use embedded SQL statements in the script to perform operations on the database. The features supported when you use embedded SQL depend on the DBMS to which your application connects.

Overview

When you use the Informix IN9 database interface to connect to a database, you can use embedded SQL in your scripts. You can embed the following types of SQL statements in scripts and user-defined functions:

- Transaction management statements
- Non-cursor statements
- Cursor statements
- Database stored procedures

Informix API

The Informix database interfaces use the Informix application programming interface (API) to interact with the database.

When you use embedded SQL, PowerBuilder makes the required calls to the API. Therefore, you do not need to know anything about the Informix API in order to use embedded SQL in PowerBuilder.

See also

Chapter 9, Using Informix
Informix transaction management statements
Informix non-cursor statements
Informix cursor statements
Informix database stored procedures
Informix name qualification

Informix name qualification

Since PowerBuilder does not inspect all SQL statement syntax, you can qualify Informix catalog entities as necessary.

For example, these qualifications are all acceptable:

- emp_name
- employee.emp_name
- Informix.employee.emp_name

Functions

You can use any function that Informix supports (such as aggregate or mathematical functions) in SQL statements.

This example illustrates how to call the Informix function HEX in a SELECT statement:

```
SELECT HEX(emp_num)
    INTO :emp_name_var
    FROM employee;DBMS=ODB
```

Informix transaction management statements

Qualification

You can use the following transaction management statements with transaction objects to manage connections and transactions for Informix databases:

For example, these qualifications are all acceptable:

- CONNECT
- COMMIT
- DISCONNECT
- ROLLBACK

See also

Informix using CONNECT, COMMIT, DISCONNECT, and ROLLBACK

Informix using CONNECT, COMMIT, DISCONNECT, and ROLLBACK

This table lists each transaction management statement and describes how it works when you use the Informix IN9 interface to connect to a database:

Statement	Description
CONNECT	Establishes the database connection. After you assign values to the required properties of the transaction object, you can execute a CONNECT. After this call completes successfully, PowerBuilder issues a BEGIN WORK to start a logical unit of work for the transaction.
COMMIT	Terminates the logical unit of work, guarantees that all changes made to the database since the beginning of the current unit of work become permanent, and starts a new logical unit of work.
DISCONNECT	Terminates a successful connection. DISCONNECT automatically executes a COMMIT to guarantee that all changes made to the database since the beginning of the current unit of work are committed.
ROLLBACK	Terminates a logical unit of work, undoes all changes made to the database since the beginning of the logical unit of work, and starts a new logical unit of work.

See also

Informix performance and locking

Informix performance and locking

An important consideration when designing a database application is deciding when connect and commit statements should occur to maximize performance and limit locking and resource use. A connect takes a certain amount of time and can tie up resources during the life of the connection. If this time is significant, then limiting the number of connects is desirable.

After a connection is established, SQL statements can cause locks to be placed on database entities. The more locks there are in place at a given moment in time, the more likely it is that the locks will hold up another transaction.

Rules

No set of rules for designing a database application is totally comprehensive. However, when you design a PowerBuilder application, you should do the following:

- Long-running connections

  Determine whether you can afford to have long-running connections. If not, your application should connect to the database only when absolutely necessary. After all the work for that connection is complete, the transaction should be disconnected.

If long-running connections are acceptable, then commits should be issued as often as possible to guarantee that all changes do in fact occur. More importantly, COMMITs should be issued to release any locks that may have been placed on database entities as a result of the statements executed using the connection.

- Background color

  SetTrans or SetTransObject function Determine whether you want to use default DataWindow transaction processing (the SetTrans function) or control the transaction in a script (the SetTransObject function).

  If you cannot afford to have long-running connections and therefore have many short-lived transactions, use the default DataWindow transaction processing. If you want to keep connections open and issue periodic COMMITs, use the SetTransObject function and control the transaction yourself.

Isolation feature

Informix-OnLine databases use the isolation feature to support assorted database lock options. In PowerBuilder, you can use the Lock property of the transaction object to set the isolation level when you connect to the database.

The following example shows how to set the Lock property to Committed read:

```
// Set the lock property to committed read
// in the default transaction object SQLCA.
SQLCA.Lock = "Committed read"
```

Informix-SE databases do not support Lock
The Lock property applies only to Informix-OnLine databases. Informix-SE (Standard Edition) databases do not support the use of lock values and isolation levels.

Example 1

This script uses embedded SQL to connect to a database and insert a row in the ORDER_HEADER table and a row in the ORDER_ITEM table. Depending on the success of the statements in the script, the script executes a COMMIT or ROLLBACK:

```
// Set the SQLCA connection properties.
SQLCA.DBMS = "IN9"
SQLCA.database = "ORDERS"// Connect to the database.

CONNECT USING SQLCA;
// Insert a row into the ORDER_HEADER table.
// A ROLLBACK is required only if the first row
// was inserted successfully.
```

```
INSERT INTO ORDER_HEADER (ORDER_ID,CUSTOMER_ID)
    VALUES ( 7891, 129 );

// Test return code for ORDER_HEADER insertion
if SQLCA.sqlcode = 0 then

// Since the ORDER_HEADER is inserted,
// try to insert ORDER_ITEM
    INSERT INTO ORDER_ITEM
        (ORDER_ID, ITEM_NBR, PART_NBR, QTY)
        VALUES ( 7891, 1, '991PLS', 456 );

// Test return code for ORDER_ITEM insertion.
    if SQLCA.sqlcode = -1 then

// If insert failed
// ROLLBACK insertion of ORDER_HEADER.
    ROLLBACK USING SQLCA;
    end if
end if

// Disconnect from the database.
DISCONNECT USING SQLCA;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

This example uses the scripts for the Open and Close events in a window and the Clicked event in a CommandButton to illustrate how you can manage transactions in a DataWindow control. Assume the window contains a DataWindow control dw_1 and the user enters data in dw_1 and then clicks the Cb_Update button to send the data to the database.

Since this script uses SetTransObject to connect to the database, the programmer is responsible for managing the transaction.

Window Open event script

```
// Set the transaction object
properties
// and connect to the database.
// Set the SQLCA connection properties.
SQLCA.DBMS = "IN9"
SQLCA.database = "ORDERS"
```

```
// Connect to the database.
CONNECT USING SQLCA;

// Tell the DataWindow which transaction object
// to use.
SetTransObject( dw_1, SQLCA )
```

CommandButton Clicked event script

```
// Declare ReturnValue an
integer.
// integerReturnValue
ReturnValue = Update( dw_1 )

// Test to see if updates were successful.
if ReturnValue = -1 then

// Updates were not successful. Since we used
// SetTransObject, rollback any changes made
// to the database.
   ROLLBACK USING SQLCA;
else


// Updates were successful. Since we used
// SetTransObject, commit any changes made
// to the database.
   COMMIT USING SQLCA;
end if
```

Window Close event script

```
// Disconnect from the database.
DISCONNECT USING SQLCA;
```

Informix non-cursor statements

The statements that do not involve cursors or stored procedures are:

- DELETE
- INSERT
- UPDATE
- SELECT (singleton)

See also

Informix DELETE, INSERT, and UPDATE
Informix SELECT

Informix DELETE, INSERT, and UPDATE

Internally, PowerBuilder processes DELETE, INSERT, and UPDATE statements the same way. PowerBuilder inspects them for any PowerScript variable references and replaces all such references with a constant that conforms to Informix rules for that data type.

Row serial number
The serial number of the row is stored in the SQLReturnData property of the transaction object after an INSERT statement executes. (The SQLReturnData property is updated after embedded SQL only; it is not updated after a DataWindow operation.)

Example

Assume you enter the following statement:

```
DELETE FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var is a PowerScript variable with the data type of integer that has been defined within the scope of the script that contains the DELETE statement. Before the DELETE statement is executed, emp_id_var is assigned a value (for example, 691) so that when the DELETE statement executes, the database receives the following statement:

```
DELETE FROM employee WHERE emp_id = 691;
```

When is this substitution technique used?
This variable substitution technique is used for all PowerScript variable types. When you use embedded SQL, precede all PowerScript variables with a colon (:).

See also

Informix SELECT

Informix SELECT

The SELECT statement contains input variables and output variables. Input variables are passed to the database as part of the execution and the substitution as described above for DELETE, INSERT, and UPDATE. Output variables are used to return values based on the result of the SELECT statement.

Example 1

Assume you enter the following statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

Here emp_id_var, emp_salary_var, and emp_name_var are PowerScript variables defined within the scope of the script that contains the SELECT statement, emp_id_var is processed as described in the DELETE example above.

Both emp_name_var and emp_salary_var are output variables that will be used to return values from the database. The data types of emp_name_var and emp_salary_var should be the PowerScript data types that best match the Informix data type. When the data types do not match perfectly, PowerBuilder converts them.

How big should numeric output variables be?
For numeric data, the output variable must be large enough to hold any value that may come from the database.

Assume the value for emp_id_var is 691 as in the previous example. When the SELECT statement executes, the database receives the following statement:

```
SELECT emp_name, emp_salary
    FROM employee WHERE emp_id = 691;
```

If no errors are returned on the execution, data locations are internally bound for the result fields. The data returned into these locations is converted if necessary and the appropriate PowerScript variables are set to those values.

Example 2

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. It also assumes the data type of the emp_id column in the employee table is CHARACTER[10].

The user enters an employee ID into the line edit sle_Emp and clicks the button Cb_Delete to delete the employee.

The script for the Clicked event in the CommandButton Cb_Delete is:

```
// Make sure we have a value.
if sle_Emp.text <> "" then

// Since we have a value, let's try to delete it.
    DELETE FROM employee WHERE emp_id = :sle_Emp.text;

// Test to see if the DELETE worked.
    if SQLCA.sqlcode = 0 then
```

```
// It seems to have worked, let user know.
MessageBox( "Delete",&
"The delete has been successfully processed!")
    else

// It didn't work.
MessageBox( "Error", &
"The delete failed. Employee ID is not valid.")
    end if
else

// No input value. Prompt user.
MessageBox( "Error", &
"An employee ID is required for "+"delete!" )
    end if
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 3

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. The user wants to extract rows from the employee table and insert them into the table named extract_employees.

The extraction occurs when the user clicks the button Cb_Extract. The boolean variable YoungWorkers is set to TRUE or FALSE elsewhere in the application.

The script for the Clicked event for the CommandButton Cb_Extract is:

```
integerEmployeeAgeLowerLimit
integerEmployeeAgeUpperLimit

// Do they have young workers?
if ( YoungWorkers = TRUE ) then

// Yes - set the age limit in the YOUNG range.
// Assume no employee is under legal working age.
    EmployeeAgeLowerLimit = 16

// Pick an upper limit.
    EmployeeAgeUpperLimit = 42
else
```

```
// No - set the age limit in the OLDER range.
   EmployeeAgeLowerLimit = 43

// Pick an upper limit that includes all
// employees.
   EmployeeAgeUpperLimit = 200
end if
INSERT INTO extract_employees(emp_id,emp_name)
   SELECT emp_id, emp_name FROM employee
      WHERE emp_age >= :EmployeeAgeLowerLimit
      AND emp_age <= :EmployeeAgeUpperLimit;
```

Informix cursor statements

In embedded SQL, statements that retrieve data and statements that update data can both involve cursors.

Retrieval statements

The retrieval statements that involve cursors are:

- DECLARE *cursor_name* CURSOR FOR . . .
- OPEN *cursor_name*
- FETCH *cursor_name* INTO . . .
- CLOSE *cursor_name*

Update statements

The update statements that involve cursors are:

- UPDATE . . . WHERE CURRENT OF *cursor_name*
- DELETE . . . WHERE CURRENT OF *cursor_name*

See also

Informix retrieval using cursors
Informix FETCH statements
Informix CLOSE for cursors

Informix retrieval using cursors

Retrieval using cursors is conceptually similar to the singleton SELECT discussed earlier. The main difference is that there can be multiple rows in a result set when you use a cursor and you control when the next row is fetched into PowerScript variables.

If you expect only a single row to exist in the employee table with the specified emp_id, use the singleton SELECT. In a singleton SELECT, you specify the SELECT statement and destination variables in one concise SQL statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

However, when a SELECT may return multiple rows, you must:

1. Declare a cursor.
2. Open it (which effectively executes the SELECT).
3. Fetch rows as needed.
4. Close the cursor.

Declaring and opening a cursor

Declaring a cursor is tightly coupled with the OPEN statement. The DECLARE specifies the SELECT statement to be executed, and the OPEN actually executes it.

Scroll cursors

When you fetch rows in an Informix database table, using a scroll cursor allows you to fetch rows in the active set in any sequence. That is, you can fetch the next row, previous row, last row, or first row.

To specify that you want to use a scroll cursor when connecting to an Informix database, set the Scroll DBParm parameter to 1. By default, PowerBuilder does not use scroll cursors in an Informix connection (the Scroll parameter is set to 0).

You cannot update scroll cursors
Scroll cursors are not updatable. If you try to declare a scroll cursor and make it updatable, it will fail.

See also

Informix nonupdatable cursors
Informix updatable cursors

Informix nonupdatable cursors

Declaring a cursor is similar to declaring a variable; a cursor is a nonexecutable statement just like a variable declaration. The first step in declaring a nonupdatable cursor is to define how the result set looks. To do this, you need a SELECT statement. You must associate the result set with a logical name so you can refer to it in subsequent SQL statements.

Example Assume the SingleLineEdit control sle_1 contains the state code for the retrieval:

The script for the Clicked event for the CommandButton Cb_Extract is:

```
// Declare cursor emp_curs for employee table.
// retrieval

DECLARE emp_curs CURSOR FOR
    SELECT emp_id, emp_name FROM Employee
    WHERE emp_state = :sle_1.text;

// Declare local variables for retrieval.
stringemp_id_var
stringemp_name_var

// Execute the SELECT statement with
// the current value of sle_1.text.
OPEN emp_curs;

// At this point, if there are no errors,
// the cursor is available for further processing.
```

Informix updatable cursors

To declare an updatable cursor, use the FOR UPDATE keywords in the declaration.

Example This statement uses the FOR UPDATE syntax to declare an updatable cursor:

```
DECLARE emp_curs CURSOR FOR
    SELECT emp_id, emp_name FROM Employee
    WHERE emp_state = :sle_1.text
    FOR UPDATE;
```

Informix FETCH statements

Qualification

The Informix database interfaces support the following FETCH statements:

- FETCH NEXT
- FETCH FIRST
- FETCH PRIOR
- FETCH LAST

See also

Informix FETCH NEXT
Informix FETCH FIRST, FETCH PRIOR, and FETCH LAST

Informix FETCH NEXT

In the singleton SELECT, you specify variables to hold the values for the columns within the selected row. The FETCH statement syntax is similar to the syntax of the singleton SELECT. Values are returned INTO a specified list of variables.

```
// Go get the first row from the result set
FETCH emp_curs INTO :emp_id_var, :emp_name_var;
```

If at least one row can be retrieved, this FETCH places the values of the emp_id and emp_name columns from the first row in the result set into the PowerScript variables emp_id_var and emp_name_var. Executing another FETCH statement will place the variables from the next row into specified variables.

FETCH statements typically occur in a loop that processes several rows from a result set (one row at a time); fetch the row, process the variables, and then fetch the next row.

What happens when the result set is exhausted?
When a result set has been exhausted, FETCH returns +100 (not found) in the SQLCode property within the referenced transaction object. This is an informational return code; -1 in SQLCode indicates an error.

See also

Informix FETCH statements
Informix FETCH FIRST, FETCH PRIOR, and FETCH LAST

Informix FETCH FIRST, FETCH PRIOR, and FETCH LAST

In addition to the conventional FETCH NEXT, the Informix interface supports FETCH FIRST, FETCH PRIOR, and FETCH LAST statements.

What if you only enter FETCH?
If you only enter FETCH, PowerBuilder assumes FETCH NEXT.

Example

This cursor example illustrates how you can loop through a result set. Assume the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed.

The statements retrieve rows from the employee table and then display a message box with the employee name in each row that is found.

```
// Declare the emp_curs
DECLARE emp_curs CURSOR FOR
    SELECT emp_name FROM EMPLOYEE
    WHERE emp_state = :sle_1.text;

// Declare a destination variable for employee
// names.
stringemp_name_var

// Get current value of sle_1.text.
OPEN emp_curs;

// Fetch the first row from the result set.
FETCH emp_curs INTO :emp_name_var;

// Loop through result set until exhausted
DO WHILE SQLCA.sqlcode = 0

// Pop up a message box with the employee name
    MessageBox("Found an employee!",emp_name_var)

// Fetch the next row from the result set
    FETCH emp_curs INTO :emp_name_var;

LOOP

// All done, so close the cursor
CLOSE emp_curs;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

See also

Informix FETCH statements
Informix FETCH NEXT

Informix CLOSE for cursors

The CLOSE statement terminates processing for the specified cursor. CLOSE releases resources associated with the cursor, and subsequent references to that cursor are allowed only if another OPEN is executed. Although you can have multiple cursors open at the same time, you should close the cursors as soon as possible for efficiency reasons.

Informix database stored procedures

Qualification

One of the most significant features of Informix is support for database stored procedures. You can use database stored procedures for:

- Retrieval only
- Update only
- Update and retrieval

PowerBuilder supports all of these uses in embedded SQL.

See also

Informix retrieval using database stored procedures
Informix update using database stored procedures
Informix database stored procedures
Informix retrieval using database stored procedures

Informix retrieval using database stored procedures

Qualification

PowerBuilder uses a construct that is very similar to cursors to support retrieval using database stored procedures. In PowerBuilder embedded SQL, there are four commands that involve database stored procedures:

- DECLARE procedure_name PROCEDURE FOR . . .
- EXECUTE PROCEDURE procedure_name
- FETCH procedure_name INTO . . .
- CLOSE procedure_nameRetrieval only

See also

Informix DECLARE and EXECUTE
Informix FETCH
Informix CLOSE

Informix DECLARE and EXECUTE

PowerBuilder requires a declarative statement to identify the database stored procedure that is being used and specify a logical name. The logical name is used to reference the procedure in subsequent SQL statements. The general syntax for declaring a procedure is:

```
DECLARE logical_procedure_name PROCEDURE FOR
    Informix_procedure_name
    ({:arg1,:arg2 , ...})
    {USING transaction_object};
```

where *logical_procedure_name* can be any valid PowerScript identifier and *Informix_procedure_name* is the name of the stored procedure in the Informix database. The parentheses after *Informix_procedure_name* are required even if the procedure has no parameters.

Creating a stored procedure

The default SQL terminator character for the Database painter is a semicolon (;). Informix also uses a semicolon in its stored procedure syntax. Therefore, to create a stored procedure in the Database painter, you must change the SQL terminator character to something other than a semicolon, such as a backquote (`).

To change the Database painter's SQL terminator character, type the character you want in the SQL Terminator Character box in the Database Preferences dialog box.

The parameter references can take the form of any valid parameter string that Informix accepts. PowerBuilder does not inspect the parameter list format except for purposes of variable substitution. The USING clause is required only if you are using a transaction object other than the default transaction object (SQLCA).

Example

Assume a stored procedure proc1 is defined as:

```
CREATE PROCEDURE proc1 AS
    SELECT emp_name FROM employee
```

To declare that procedure for processing within PowerBuilder, enter:

```
DECLARE emp_proc PROCEDURE FOR proc1;
```

Note that this declaration is a nonexecutable statement, just like a cursor declaration. Where cursors have an OPEN statement, procedures have an EXECUTE statement.

When an EXECUTE statement is executed, the procedure is invoked. The EXECUTE refers to the logical procedure name:

```
EXECUTE emp_proc;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Issuing EXECUTE statements

Use PowerBuilder embedded SQL syntax when you enter an embedded EXECUTE statement in a script; do not enter the PROCEDURE keyword. Use this syntax:

```
EXECUTE procedure_name;
```

Specify the EXECUTE statement the same way whether or not a stored procedure takes arguments. The arguments used in the DECLARE statement get passed automatically, without your having to state them in the EXECUTE statement.

Informix FETCH

To access rows returned in a result set, you use the FETCH statement the same way you use it for cursors. The FETCH statement can be executed after any EXECUTE statement that refers to a procedure that returns a result set.

Informix syntax
PowerBuilder supports Informix syntax; however, the default syntax displayed in the DataWindow painter is the most general syntax. You can leave the syntax unchanged or edit the displayed syntax to conform to the Informix syntax rules. If you do not change the syntax, PowerBuilder converts it to Informix syntax before passing it to the Informix database.

Example 1

```
FETCH emp_proc INTO :emp_name_var;
```

You can use this FETCH statement only to access values produced with a SELECT statement in a database stored procedure. You cannot use the FETCH statement to access computed rows.

The result sets that will be returned when a database stored procedure executes cannot be determined at compile time. Therefore, you must code FETCH statements that exactly match the format of a result set returned by the stored procedure when it executes.

Example 2

Assume you changed the second fetch statement in the preceding statement to:

```
FETCH emp_proc2
    INTO :part_var1,:part_var2,:part_var3;
```

The code would compile without errors, but an execution error would occur because the number of columns in the FETCH statement does not match the number of columns in the current result set. The second result set returns values from only one column.

Informix CLOSE

If a database stored procedure returns a result set, it must be closed when processing is complete.

Do you have to retrieve all the rows?
You do not have to retrieve all rows in a result set to close a request or procedure.

Closing a procedure looks the same as closing a cursor:

```
CLOSE emp_proc;
```

As with cursors, if a procedure executes successfully and returns at least one result set and is not closed, a result set is pending and no SQL commands other than the FETCH can be executed. Procedures with result sets should be closed as soon as possible.

The procedure remains open until you close it, execute a COMMIT or a ROLLBACK, or end the database connection.

Informix update using database stored procedures

Database stored procedures that only perform updates and do not return a result set are handled in much the same way as procedures that return a result set. The only difference is that after the EXECUTE procedure_name statement executes, no result set is pending and no CLOSE statement is required.

Using the SQLCode property

If you know that a particular procedure can never return a result set, only the EXECUTE statement is required. If there is a procedure that may or may not return a result set, you can test the SQLCode property of the referenced transaction object for +100 (the code for not found) after the EXECUTE.

This table shows all possible values for SQLCode after an EXECUTE:

Return code	Means
0	The EXECUTE PROCEDURE was successful and a result set is pending. Regardless of the number of FETCH statements executed, the procedure must be explicitly closed with a CLOSE statement. This code is returned even if the result set is empty.
+100	Fetched row not found
-1	The EXECUTE was not successful and no result set was returned.

Example 1

This example illustrates how to execute a database stored procedure that does not return a result set:

```
// good_employee is an Informix stored procedure.
// Declare the procedure.
DECLARE good_emp_p 1roc PROCEDURE FOR good_employee;
EXECUTE good_emp_proc;

// Test return code. Allow for +100 since you do
// not expect a result set.
if SQLCA.sqlcode = -1 then

// Issue error message since it failed.
   MessageBox("Stored Procedure Error!", &
   SQLCA.sqlerrtext)
end if
```

Example 2

This example illustrates how to pass parameters to a database stored procedure that returns a result set. Emp_id_var has been set elsewhere to 691:

```
// Get_employee is an Informix stored procedure.
// Declare the procedure.
DECLARE get_emp_proc PROCEDURE FOR
   get_employee @emp_id_parm = :emp_id_var;

// Declare a destination variable for emp_name
stringemp_name_var

// Execute the stored procedure using the
// current value for emp_id_var.
EXECUTE get_emp_proc;

// Test return code to see if it worked.
if SQLCA.sqlcode = 0 then

// We got a row, so fetch it and display it.
   FETCH get_emp_proc INTO :emp_name_var;

// Display the employee name.
   MessageBox("Got my employee!",emp_name_var)

// You are all done, close the procedure.
   CLOSE Get_emp_proc;
end if
```

Informix using database stored procedures in DataWindow objects

You can use database stored procedures as a data source for DataWindow objects. The following considerations apply:

- Result set definition

  You must define what the result set looks like. The DataWindow object cannot determine this information from the stored procedure definition in the database.

- DataWindow updates

  You cannot perform DataWindow updates through stored procedures (that is, you cannot update the database with changes made in the DataWindow object); only retrieval is allowed. (However, the DataWindow can have update characteristics set manually through the DataWindow painter.)

- Result set processing

  You can specify only one result set to be processed when you define the stored procedure result set in the DataWindow painter.

- Computed rows

  Computed rows cannot be processed in DataWindows.

- Informix syntax

  PowerBuilder supports Informix syntax; however, the syntax displayed in the DataWindow painter is the most general syntax. You can leave the syntax unchanged or edit the displayed syntax to conform to the Informix syntax rules. If you do not change the syntax, PowerBuilder converts it to Informix syntax before passing it to the Informix database.

Informix database stored procedure summary

When you use database stored procedures in a PowerBuilder application, keep the following points in mind:

- Manipulating stored procedures

  To manipulate database stored procedures, PowerBuilder provides SQL statements that are similar to cursor statements.

- Retrieval and update

  PowerBuilder supports retrieval, update, or a combination of retrieval and update in database stored procedures, including procedures that do not return a result set and those that return a result set.

- Transactions and procedures without result sets

  When a procedure executes using a particular connection (transaction) and the procedure does not return a result set, the procedure is no longer active. No result set is pending and, therefore, you do not execute a CLOSE statement.

CHAPTER 22 Using Embedded SQL with Microsoft SQL Server

About this chapter

When you create scripts for a PowerBuilder application, you can use embedded SQL statements in the script to perform operations on the database. The features supported when you use embedded SQL depend on the DBMS to which your application connects.

Overview

When your PowerBuilder application connects to a SQL Server database, you can use embedded SQL in your scripts. This interface uses the DB-Library (DB-Lib) client API to access the database.

When you use the SQL Server database interface, you can embed the following types of SQL statements in scripts and user-defined functions:

- Transaction management statements
- Non-cursor statements
- Cursor statements
- Database stored procedures

DB-Library API

The Microsoft SQL Server database interface uses the DB-Library (DB-Lib) application programming interface (API) to access the database. When you use embedded SQL, PowerBuilder makes the required calls to the API. Therefore, you do not need to know anything about DB-Lib to use embedded SQL in PowerBuilder.

See also

Microsoft SQL Server Functions
Microsoft SQL Server Transaction management statements
Microsoft SQL Server Non-cursor statements
Microsoft SQL Server Cursor statements
Microsoft SQL Server Using database stored procedures in DataWindow objects
Microsoft SQL Server Name qualification

Microsoft SQL Server Name qualification

Since PowerBuilder does not inspect all SQL statement syntax, you can qualify SQL Server catalog entities as necessary.

For example, the following qualifications are all acceptable:

- emp_name
- employee.emp_name
- dbo.employee.emp_name
- emp_db.dbo.employee.emp_name

Microsoft SQL Server Functions

You can use any function that SQL Server supports (such as aggregate or mathematical functions) in SQL statements.

This example shows how to use the SQL Server function UPPER in a SELECT statement:

```
SELECT UPPER(emp_name)
    INTO :emp_name_var
    FROM employee;
```

Calling DB-Library functions

While PowerBuilder provides access to a large percentage of the features within SQL Server, in some cases you may decide that you need to call one or more DB-Lib functions directly for a particular application. PowerBuilder provides access to any Windows DLL by using external function declarations.

The DB-Lib calls qualify for this type of access. Most DB-Lib calls require a pointer to a DBPROCESS structure as their first parameter. If you want to call DB-Lib without reconnecting to the database to get a DBPROCESS pointer, use the PowerScript DBHandle function.

DBHandle

DBHandle takes a transaction object as a parameter and returns a long variable, which is the handle to the database for the transaction. This handle is actually the DBPROCESS pointer that PowerBuilder uses internally to communicate with the database. You can use this returned long value in the SQL Server DLLs and pass it as one of the parameters in your function.

This example shows how to use DBHandle. Assume a successful connection has occurred using the default transaction object (SQLCA):

```
// Define a variable to hold our DB handle.
long    SQLServerHandle

// Go get the handle.
SQLServerHandle = SQLCA.DBHandle( )

// Now that you have the DBPROCESS pointer,
// call the DLL function.
MyDLLFunction( SQLServerHandle, parm1, parm2, ... )
```

In your DLL, cast the incoming long value into a pointer to a DBPROCESS structure:

```
MyDLLFunction( long 1SQLServerHandle,
      parm1_type parm1,
      parm2_type Parm2, ... )

{
DBPROCESS * pDatabase;
pDatabase = (DBPROCESS *)  1SQLServerHandle;
// DB-Lib functions can be called using pDatabase.
}
```

Microsoft SQL Server Transaction management statements

Transaction management statements

You use the following transaction management statements with transaction objects to manage connection and transactions for a SQL Server database:

- CONNECT
- COMMIT
- DISCONNECT
- ROLLBACK

Transaction management in triggers

You should not use transaction statements in triggers. A *trigger* is a special kind of stored procedure that takes effect when you issue a statement such as INSERT, DELETE, or UPDATE on a specified table or column. Triggers can be used to enforce referential integrity.

For example, assume that a certain condition within a trigger is not met and you want to execute a ROLLBACK. Instead of coding the ROLLBACK directly in the trigger, you should use RAISERROR and test for that particular return code in the DBMS-specific return code (SQLDBCode) property within the referenced transaction object.

See also

Microsoft SQL Server Using CONNECT, COMMIT, DISCONNECT, and ROLLBACK

Microsoft SQL Server Using CONNECT, COMMIT, DISCONNECT, and ROLLBACK

The following table lists each transaction management statement and describes how it works when you use the SQL Server interface to connect to a database:

Statement	Description
CONNECT	Establishes the database connection. After you assign values to the required properties of the transaction object, you can execute a CONNECT. After the CONNECT completes successfully, PowerBuilder automatically starts a SQL Server transaction. This is the start of a logical unit of work.
COMMIT	COMMIT terminates the logical unit of work, guarantees that all changes made to the database since the beginning of the current unit of work become permanent, and starts a new logical unit of work. If AutoCommit is false, a COMMIT TRANSACTION executes, then a BEGIN TRANSACTION executes to start a new logical unit of work. If AutoCommit is true, an error occurs when a COMMIT executes.
DISCONNECT	Terminates a successful connection. DISCONNECT automatically executes a COMMIT to guarantee that all changes made to the database since the beginning of the current unit of work are committed. If AutoCommit is false, a COMMIT TRANSACTION executes automatically to guarantee that all changes made to the database since the beginning of the current logical unit of work are committed.

Statement	Description
ROLLBACK	ROLLBACK terminates a logical unit of work, undoes all changes made to the database since the beginning of the logical unit of work, and starts a new logical unit of work. If AutoCommit is false, a ROLLBACK TRANSACTION executes, then a BEGIN TRANSACTION executes to start a new logical unit of work. If AutoCommit is true, an error occurs when a ROLLBACK executes.

See also

Microsoft SQL Server Performance and locking
Microsoft SQL Server Temporary tables
Microsoft SQL Server Using AutoCommit

Microsoft SQL Server Using AutoCommit

Using AutoCommit

The setting of the AutoCommit property of the transaction object determines whether PowerBuilder issues SQL statements inside or outside the scope of a transaction. When AutoCommit is set to false or 0 (the default), SQL statements are issued inside the scope of a transaction. When you set AutoCommit to true or 1, SQL statements are issued outside the scope of a transaction.

Versions of SQL Server prior to SQL Server 2000 require you to execute Data Definition Language (DDL) statements outside the scope of a transaction. If you execute a database stored procedure that contains DDL statements within the scope of a transaction, an error message is returned and the DDL statements are rejected. When you use the transaction object to execute a database stored procedure that creates a temporary table, you do not want to associate the connection with a transaction.

To execute SQL Server stored procedures containing DDL statements in SQL Server 7 and earlier, you must set AutoCommit to true so PowerBuilder issues the statements outside the scope of a transaction. However, if AutoCommit is set to true, you cannot issue a ROLLBACK. Therefore, you should set AutoCommit back to false (the default) immediately after completing the DDL operation.

When you change the value of AutoCommit from false to true, PowerBuilder issues a COMMIT statement by default.

See also

Microsoft SQL Server Performance and locking
Microsoft SQL Server Temporary tables

Microsoft SQL Server Using CONNECT, COMMIT, DISCONNECT, and ROLLBACK

Microsoft SQL Server Performance and locking

An important consideration when designing a database application is deciding when CONNECT and COMMIT statements should occur to maximize performance and limit locking and resource use. A CONNECT takes a certain amount of time and can tie up resources during the life of the connection. If this time is significant, then limiting the number of CONNECT statements is desirable.

In addition, after a connection is established, SQL statements can cause locks to be placed on database entities. The more locks at a given moment in time, the more likely it is that the locks will hold up another transaction.

Rules

No set of rules for designing a database application is totally comprehensive. However, when you design a PowerBuilder application, you should do the following:

- **Long-running connections** Determine whether you can afford to have long-running connections. If not, your application should connect to the database only when absolutely necessary. After all the work for that connection is complete, the transaction should be disconnected.

  If long-running connections are acceptable, then COMMITs should be issued as often as possible to guarantee that all changes do in fact occur. More importantly, COMMITs should be issued to release any locks that may have been placed on database entities as a result of the statements executed using the connection.

- **SetTrans or SetTransObject function** Determine whether you want to use default DataWindow transaction processing (the SetTrans function) or control the transaction in a script (the SetTransObject function).

  If you cannot afford to have long-running connections and therefore have many short-lived transactions, use the default DataWindow transaction processing. If you want to keep connections open and issue periodic COMMITs, use the SetTransObject function and control the transaction yourself.

Isolation feature

SQL Server uses the isolation feature to support assorted database lock options. In PowerBuilder, you can use the Lock property of the transaction object to set the isolation level when you connect to a SQL Server database.

The following example shows how to set the Lock property to RU (Read uncommitted):

```
// Set the lock property to read uncommitted
// in the default transaction object SQLCA.
SQLCA.Lock = "RU"
```

Example 1

This script uses embedded SQL to connect to a database and insert a row in the ORDER_HEADER table and a row in the ORDER_ITEM table. Depending on the success of the statements in the script, the script executes either a COMMIT or a ROLLBACK.

```
// Set the SQLCA connection properties.
SQLCA.DBMS = "SQLServer"
SQLCA.servername = "SERVER24"
SQLCA.database = "ORDERS"
SQLCA.logid = "JPL"
SQLCA.logpass = "TREESTUMP"

// Connect to the database.
CONNECT USING SQLCA;

// Insert a row into the ORDER_HEADER table.
// A ROLLBACK is required only if the first row
// was inserted successfully.
INSERT INTO ORDER_HEADER (ORDER_ID,CUSTOMER_ID)
    VALUES ( 7891, 129 );

// Test return code for ORDER_HEADER insertion.
if SQLCA.sqlcode = 0 then

// Since the ORDER_HEADER is inserted,
// try to insert ORDER_ITEM.
INSERT INTO ORDER_ITEM(ORDER_ID, ITEM_NBR,
    PART_NBR, QTY)
       VALUES ( 7891, 1, '991PLS', 456 );

// Test return code for ORDER_ITEM insertion.
    if SQLCA.sqlcode = -1 then

// If insert failed.
// ROLLBACK insertion of ORDER_HEADER.
    ROLLBACK USING SQLCA;
    end if
end if

// Commit changes and disconnect from the database.
```

```
DISCONNECT USING SQLCA;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

This example uses the scripts for the Open and Close events in a window and the Clicked event in a CommandButton to illustrate how you can manage transactions in a DataWindow control. Assume that the window contains a DataWindow control dw_1 and that the user enters data in dw_1 and then clicks the Cb_Update button to send the data to the database.

Since this script uses SetTransObject to connect to the database, the programmer is responsible for managing the transaction.

The window OPEN event script:

```
// Set the transaction object properties
// and connect to the database.
// Set the SQLCA connection properties.
SQLCA.DBMS = "SQLServer"
SQLCA.servername = "SERVER24"
SQLCA.database = "ORDERS"
SQLCA.logid = "JPL"
SQLCA.logpass = "TREESTUMP"

// Connect to the database.
CONNECT USING SQLCA;

// Tell the DataWindow which transaction object
// to use.
SetTransObject( dw_1, SQLCA )
```

The CommandButton CLICKED event script:

```
// Declare ReturnValue an integer.
integer ReturnValue
ReturnValue = Update( dw_1 )

// Test to see if updates were successful.
if ReturnValue = -1 then

// Updates were not successful. Since we used
// SetTransObject, rollback any changes made
// to the database.
   ROLLBACK USING SQLCA;
else
```

```
// Updates were successful. Since we used
// SetTransObject, commit any changes made
// to the database.
    COMMIT USING SQLCA;
end if
```

The window CLOSE event script:

```
// Disconnect from the database.
DISCONNECT USING SQLCA;
```

See also

Microsoft SQL Server Temporary tables

Microsoft SQL Server Non-cursor statements

The statements that do not involve cursors or procedures are:

- DELETE (Microsoft SQL Server DELETE, INSERT, and UPDATE)
- INSERT (Microsoft SQL Server DELETE, INSERT, and UPDATE)
- SELECT (Microsoft SQL Server SELECT) (singleton)
- UPDATE (Microsoft SQL Server DELETE, INSERT, and UPDATE)

Microsoft SQL Server DELETE, INSERT, and UPDATE

Internally, PowerBuilder processes DELETE, INSERT, and UPDATE statements the same way. PowerBuilder inspects them for any PowerScript data variable references and replaces all such references with a constant that conforms to SQL Server rules for the data type.

Example

Assume you enter the following statement:

```
DELETE FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var is a PowerScript data variable with the data type of integer that has been defined within the scope of the script that contains the DELETE statement. Before the DELETE statement is executed, emp_id_var is assigned a value (say 691) so that when the DELETE statement executes, the database receives the following statement:

```
DELETE FROM employee WHERE emp_id = 691;
```

When is this substitution technique used?

This variable substitution technique is used for all PowerScript variable types. When you use embedded SQL, precede all PowerScript variables with a colon (:).

See also

Microsoft SQL Server SELECT

Microsoft SQL Server SELECT

The SELECT statement contains input variables and output variables.

- Input variables are passed to the database as part of the execution and the substitution as described above for DELETE, INSERT, and UPDATE.
- Output variables are used to return values based on the result of the SELECT statement.

Example 1

Assume you enter the following statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var, emp_salary_var, and emp_name_var are variables defined within the scope of the script that contains the SELECT statement, and emp_id_var is processed as described in the DELETE example above.

Both emp_name_var and emp_salary_var are output variables that will be used to return values from the database. The data types of emp_name_var and emp_salary_var should be the PowerScript data types that best match the SQL Server data type. When the data types do not match perfectly, PowerBuilder converts them.

How big should numeric output variables be?
For numeric data, the output variable must be large enough to hold any value that may come from the database.

Assume the value for emp_id_var is 691 as in the previous example. When the SELECT statement executes, the database receives the following statement:

```
SELECT emp_name, emp_salary
    FROM employee WHERE emp_id = 691;
```

If the statement executes with no errors, data locations for the result fields are bound internally. The data returned into these locations is then converted as necessary and the appropriate PowerScript data variables are set to those values.

Example 2

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. It also assumes the data type of the emp_id column in the employee table is CHARACTER[10].

The user enters an employee ID into the single line edit field sle_Emp and clicks the button Cb_Delete to delete the employee.

The script for the Clicked event in the CommandButton Cb_Delete is:

```
// Make sure we have a value.
if sle_Emp.text <> "" then

// Since we have a value, try to delete it.
    DELETE FROM employee
    WHERE emp_id = :sle_Emp.text;

// Test to see if the DELETE worked.
    if SQLCA.sqlcode = 0 then

// It seems to have worked, let user know.
        MessageBox( "Delete",&
        "The delete has been successfully "&
        +" processed!")
    else

//It didn't work.
        MessageBox( "Error", &
         "The delete failed. Employee ID is not "&
          +"valid.")
    end if
else

// No input value. Prompt user.
        MessageBox( "Error",&
        "An employee ID is required for "&
        +"delete!" )
    end if
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 3

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. The user wants to extract rows from the employee table and insert them into the table named extract_employees. The extraction occurs when the user clicks the button Cb_Extract. The boolean variable YoungWorkers is set to TRUE or FALSE elsewhere in the application.

The script for the Clicked event for the CommandButton Cb_Extract is:

```
integer    EmployeeAgeLowerLimit
integer    EmployeeAgeUpperLimit

// Do they have young workers?
if ( YoungWorkers = TRUE ) then

// Yes - set the age limit in the YOUNG range.
// Assume no employee is under legal working age.
   EmployeeAgeLowerLimit = 16

// Pick an upper limit.
   EmployeeAgeUpperLimit = 42
else

// No - set the age limit in the OLDER range.
   EmployeeAgeLowerLimit = 43

// Pick an upper limit that includes all
// employees.
   EmployeeAgeUpperLimit = 200
end if
INSERT INTO extract_employee(emp_id,emp_name)
   SELECT emp_id, emp_name FROM employee
      WHERE emp_age >= :EmployeeAgeLowerLimit
      AND emp_age <= :EmployeeAgeUpperLimit;
```

See also

Microsoft SQL Server DELETE, INSERT, and UPDATE

Microsoft SQL Server Cursor statements

In embedded SQL, statements that retrieve data can involve cursors. These statements are:

- DECLARE *cursor_name* CURSOR FOR ...
- OPEN *cursor_name*
- FETCH *cursor_name* INTO ...

- CLOSE *cursor_name*

Note `UPDATE ... WHERE CURRENT OF` ***`cursor_name`*** and `DELETE ... WHERE CURRENT OF` ***`cursor_name`*** are not supported in SQL Server.

Retrieval

Retrieval using cursors is conceptually similar to retrieval in the singleton SELECT. The main difference is that since there can be multiple rows in a result set, you control when the next row is fetched into the PowerScript data variables.

If you expect only a single row to exist in the employee table with the specified emp_id, use the singleton SELECT. In a singleton SELECT, you specify the SELECT statement and destination variables in one concise SQL statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

However, when a SELECT may return multiple rows, you must:

1 Declare a cursor.

2 Open it (which conceptually executes the SELECT).

3 Fetch rows as needed.

4 Close the cursor.

Declaring and opening a cursor

Declaring a cursor is tightly coupled with the OPEN statement. The DECLARE specifies the SELECT statement to be executed, and the OPEN actually executes it.

Declaring a cursor is similar to declaring a variable; a cursor is a nonexecutable statement just like a variable declaration. The first step in declaring a cursor is to define how the result set looks. To do this, you need a SELECT statement. Since you must refer to the result set in subsequent SQL statements, you must associate the result set with a logical name.

Scrolling and locking

Use the CursorScroll and CursorLock DBParm parameters to specify the scrolling and locking options.

Example

Assume the SingleLineEdit sle_1 contains the state code for the retrieval:

```
// Declare cursor emp_curs for employee table
// retrieval.
DECLARE emp_curs CURSOR FOR
    SELECT emp_id, emp_name FROM EMPLOYEE
    WHERE emp_state = :sle_1.text;
```

```
// Declare local variables for retrieval.
string emp_id_var
string emp_name_var

// Execute the SELECT statement with
// the current value of sle_1.text.
OPEN emp_curs;

// At this point, if there are no errors,
// the cursor is available for further
// processing.
```

See also

Microsoft SQL Server Fetching rows
Microsoft SQL Server Closing the cursor

Microsoft SQL Server Fetching rows

The SQL Server interfaces support the following FETCH statements (Microsoft_SQL_Server_FETCH):

- FETCH NEXT (Microsoft SQL Server FETCH NEXT)
- FETCH FIRST (Microsoft SQL Server FETCH FIRST, FETCH PRIOR, and FETCH LAST)
- FETCH PRIOR (Microsoft SQL Server FETCH FIRST, FETCH PRIOR, and FETCH LAST)
- FETCH LAST (Microsoft SQL Server FETCH FIRST, FETCH PRIOR, and FETCH LAST)

Microsoft SQL Server FETCH NEXT

In the singleton SELECT, you specify variables to hold the values for the columns within the selected row. The FETCH statement syntax is similar to the syntax of the singleton SELECT. Values are returned INTO a specified list of variables.

This example continues the previous example by retrieving some data:

```
// Go get the first row from the result set.
FETCH emp_curs INTO :emp_id_var, :emp_name_var;
```

If at least one row can be retrieved, this FETCH places the values of the emp_id and emp_name columns from the first row in the result set into the PowerScript data variables emp_id_var and emp_name_var. Executing another FETCH statement will place the variables from the next row into specified variables.

FETCH statements typically occur in a loop that processes several rows from a result set (one row at a time): fetch the row, process the variables, and then fetch the next row.

What happens when the result set is exhausted?

FETCH returns +100 (not found) in the SQLCode property within the referenced transaction object. This is an informational return code; -1 in SQLCode indicates an error.

See also

Microsoft SQL Server FETCH NEXT
Microsoft SQL Server FETCH FIRST, FETCH PRIOR, and FETCH LAST

Microsoft SQL Server FETCH FIRST, FETCH PRIOR, and FETCH LAST

SQL Server support the FETCH FIRST, FETCH PRIOR, and FETCH LAST statements in addition to the conventional FETCH NEXT statement.

What if you only enter FETCH?
If you only enter FETCH, PowerBuilder assumes FETCH NEXT.

Example

This cursor example illustrates how you can loop through a result set. Assume the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed.

The statements retrieve rows from the employee table and then display a message box with the employee name in each row that is found.

```
// Declare the emp_curs.
DECLARE emp_curs CURSOR FOR
    SELECT emp_name FROM EMPLOYEE
    WHERE emp_state = :sle_1.text;

// Declare a destination variable for employee
// names.
string    emp_name_var

// Execute the SELECT statement with the
// current value of sle_1.text.
OPEN emp_curs;

// Fetch the first row from the result set.
FETCH emp_curs INTO :emp_name_var;

// Loop through result set until exhausted.
DO WHILE sqlca.sqlcode = 0

// Pop up a message box with the employee name.
   MessageBox("Found an employee!",emp_name_var)

// Fetch the next row from the result set.
   FETCH emp_curs INTO :emp_name_var;
LOOP

// All done, so close the cursor.
CLOSE emp_curs;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

See also

Microsoft SQL Server FETCH
Microsoft SQL Server FETCH NEXT

Microsoft SQL Server Closing the cursor

The CLOSE statement terminates processing for the specified cursor. CLOSE releases resources associated with the cursor, and subsequent references to that cursor are allowed only if another OPEN is executed. Although you can have multiple cursors open at the same time, you should close the cursors as soon as possible for efficiency reasons.

In SQL Server, there is an additional reason to close cursors as soon as possible. When an OPEN statement completes successfully, there is a result pending for the current connection. FETCH statements can be executed as long as there are rows in the result set to be processed. However, as long as the result set is pending, no other commands can be executed using the connection. To execute other commands using the connection, you must release the result set by closing the cursor.

Internally, PowerBuilder issues a DB-Lib dbcancel statement when the cursor is closed. After the CLOSE has been executed, the connection can be used for other SQL statements.

Example

This example illustrates the pending result set problem in SQL Server. These statements use the cursor emp_curs to retrieve rows from the employee table, then attempt to execute another SQL statement while the cursor is open:

```
// Declare the emp_curs.
DECLARE emp_curs CURSOR FOR
    SELECT emp_name FROM EMPLOYEE
    WHERE emp_state = :sle_1.text;

// Declare a destination variable for employee
// names.
string    emp_name_var

// Execute the SELECT statement with the current
// value of sle_1.text.
OPEN emp_curs;

// Execute an INSERT statement.
INSERT INTO office ( office_id, office_city )
    VALUES ( 1234, 'Boston' );

// This INSERT statement would fail because of
// the pending result set from the emp_curs
// cursor. If we had never opened the cursor, or
// if we had completed processing of the cursor
// and then closed it, the INSERT statement
// would work.
```

Microsoft SQL Server Database stored procedures

Retrieval and update

One of the most significant features of SQL Server is database stored procedures. You can use database stored procedures for:

- Retrieval only
- Update only
- Update and retrieval

PowerBuilder supports all these uses in PowerBuilder embedded SQL.

Using AutoCommit with database stored procedures

Database stored procedures often create temporary table that hold rows accumulated during processing. To create these tables, the stored procedure executes SQL Data Definition Language (DDL) statements. Versions of SQL Server prior to SQL Server 2000 do not allow you to execute DDL statements within the scope of a transaction.

To execute SQL Server stored procedures that contain DDL statements statements in SQL Server 7 and earlier, you must set the AutoCommit property of the transaction object to true so PowerBuilder issues the statements outside the scope of a transaction. However, if AutoCommit is set to true, you cannot issue a ROLLBACK. Therefore, you should set AutoCommit back to false (the default) immediately after completing the DDL operation.

When you change the value of AutoCommit from false to true, PowerBuilder issues a COMMIT statement by default.

System database stored procedures

You can access system database stored procedures the same way you access user-defined stored procedures. You can use the DECLARE statement against any procedure and can qualify procedure names if necessary.

See also

Microsoft SQL Server Retrieval
Microsoft SQL Server Temporary tables
Microsoft SQL Server Update
Microsoft SQL Server Using database stored procedures in DataWindow objects
Microsoft SQL Server Database stored procedures summary

Microsoft SQL Server Retrieval

PowerBuilder uses a construct that is very similar to cursors to support retrieval using database stored procedures. In the PowerBuilder-supported embedded SQL, there are four commands that involve database stored procedures:

- DECLARE *procedure_name* PROCEDURE FOR ...
- EXECUTE *procedure_name*
- FETCH *procedure_name* INTO ...
- CLOSE *procedure_name*

See also

Microsoft SQL Server DECLARE and EXECUTE
Microsoft SQL Server FETCH
Microsoft SQL Server CLOSE

Microsoft SQL Server DECLARE and EXECUTE

PowerBuilder requires a declarative statement to identify the database stored procedure that is being used and a logical name that can be referenced in subsequent SQL statements.

The general syntax for declaring a procedure is:

```
DECLARE logical_procedure_name PROCEDURE FOR
    SQL_Server_procedure_name
    @Param1 = value1, @Param2 = value2,
    @Param3 = value3 OUTPUT,
    {USING transaction_object} ;
```

where *logical_procedure_name* can be any valid PowerScript data identifier and *SQL_Server_procedure_name* is the name of the stored procedure in the database.

The parameter references can take the form of any valid parameter string that SQL Server accepts. PowerBuilder does not inspect the parameter list format except for purposes of variable substitution. You must use the reserved word OUTPUT to indicate an output parameter. The USING clause is required only if you are using a transaction object other than the default transaction object (SQLCA).

Example 1

Assume a stored procedure proc1 is defined as:

```
CREATE PROCEDURE proc1 AS
    SELECT emp_name FROM employee
```

To declare that procedure for processing within PowerBuilder, enter:

```
DECLARE emp_proc PROCEDURE FOR proc1;
```

Note that this declaration is a nonexecutable statement, just like a cursor declaration. Where cursors have an OPEN statement, procedures have an EXECUTE statement.

When an EXECUTE statement executes, the procedure is invoked. The EXECUTE refers to the logical procedure name:

```
EXECUTE emp_proc;
```

Example 2

To declare a procedure with input and output parameters, enter:

```
DECLARE sp_duration PROCEDURE FOR pr_date_diff_prd_ken
    @var_date_1 = :ad_start,
    @var_date_2 = :ad_end,
    @rtn_diff_prd = :ls_duration OUTPUT;
```

Microsoft SQL Server FETCH

To access rows returned in a result set, you use the FETCH statement the same way you use it for cursors. The FETCH statement can be executed after any EXECUTE statement that refers to a procedure that returns a result set.

Example 1

```
FETCH emp_proc INTO :emp_name_var;
```

You can use this FETCH statement only to access values produced with a SELECT statement in a database stored procedure. You cannot use the FETCH statement to access computed rows.

Database stored procedures can return multiple result sets. Assume you define a database stored procedure proc2 as follows:

```
CREATE PROCEDURE proc2 AS
    SELECT emp_name FROM employee
    SELECT part_name FROM parts
```

PowerBuilder provides access to both result sets:

```
// Declare the procedure.
DECLARE emp_proc2 PROCEDURE FOR proc2;

// Declare some variables to hold results.
string    emp_name_var
string    part_name_var

// Execute the stored procedure.
EXECUTE emp_proc2;

// Fetch the first row from the first result
// set.
FETCH emp_proc2 INTO :emp_name_var;

// Loop through all rows in the first result
// set.
DO WHILE sqlca.sqlcode = 0

// Fetch the next row from the first result set.
    FETCH emp_proc2 INTO :emp_name_var;
LOOP

// At this point we have exhausted the first
// result set. After this occurs,
// PowerBuilder notes that there is another
// result set and internally shifts result sets.
// The next FETCH executed will retrieve the
// first row from the second result set.
// Fetch the first row from the second result
// set.
```

```
    FETCH emp_proc2 INTO :part_name_var;
// Loop through all rows in the second result
// set.
DO WHILE sqlca.sqlcode = 0
// Fetch the next row from the second result
// set.
    FETCH emp_proc2 INTO :part_name_var;
LOOP
```

The result sets that will be returned when a database stored procedure executes cannot be determined at compile time. Therefore, you must code FETCH statements that exactly match the format of a result set returned by the stored procedure when it executes.

Example 2

In the preceding example, if instead of coding the second fetch statement as:

```
FETCH emp_proc2 INTO :part_name_var;
```

you coded it as:

```
FETCH emp_proc2
    INTO :part_var1,:part_var2,:part_var3;
```

the statement would compile without errors. But an execution error would occur: the number of columns in the FETCH statement does not match the number of columns in the current result set. The second result set returns values from only one column.

See also

Microsoft SQL Server FETCH NEXT
Microsoft SQL Server FETCH FIRST, FETCH PRIOR, and FETCH LAST

Microsoft SQL Server CLOSE

If a database stored procedure returns a result set, it must be closed when processing is complete.

Closing a procedure looks the same as closing a cursor:

```
CLOSE emp_proc;
```

As with cursors, if a procedure executes successfully and returns at least one result set and is not closed, a result set is pending and no SQL commands other than the FETCH can be executed. Procedures with result sets should be closed as soon as possible.

You do not have to retrieve all the rows in a result set to close a request or procedure.

Microsoft SQL Server Update

Using the SQL Code property

If you know for sure that a particular procedure can never return result sets, then the EXECUTE statement is all that is needed. If there is a procedure that may or may not return a result set, you can test the SQLCode property of the referenced transaction object for +100 (the code for NOT FOUND) after the EXECUTE.

The following table shows all the possible values for SQLCode after an EXECUTE:

Return code	Means
0	The EXECUTE was successful and at least one result set is pending. Regardless of the number of FETCH statements executed, the procedure must be explicitly closed with a CLOSE statement. This code is returned even if the result set is empty.
+100	Fetched row not found.
-1	The EXECUTE was not successful and no result sets were returned. The procedure does not require a CLOSE. If a CLOSE is attempted against this procedure an error will be returned.

Example 1

Assume the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed. Also assume the description of the SQL Server procedure good_employee is:

```
// SQL Server good_employee stored procedure:
CREATE PROCEDURE good_employee AS
    UPDATE employee
    SET emp_salary=emp_salary * 1.1
    WHERE emp_status = 'EXC'
```

This example illustrates how to execute a stored procedure that does not return any result sets:

```
// Declare the procedure.
DECLARE good_emp_proc PROCEDURE
FOR good_employee;

// Execute it.
```

```
EXECUTE good_emp_proc;

// Test return code. Allow for +100 since you do
// not expect result sets.
if SQLCA.sqlcode = -1 then

// Issue error message since it failed.
    MessageBox("Stored Procedure Error!", &
    SQLCA.sqlerrtext)
end if
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

Assume the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed. Also assume the description of the SQL Server procedure get_employee is:

```
// SQL Server get_employee stored procedure:
    CREATE PROCEDURE get_employee @emp_id_parm
    int AS SELECT emp_name FROM employee
    WHERE emp_id = @emp_id_parm
```

This example illustrates how to pass parameters to a database stored procedure. Emp_id_var has been set elsewhere to 691:

```
// Declare the procedure.
DECLARE get_emp_proc PROCEDURE FOR
    get_employee @emp_id_parm = :emp_id_var;

// Declare a destination variable for emp_name.
string    emp_name_var

// Execute the stored procedure using the
// current value for emp_id_var.
EXECUTE get_emp_proc;

// Test return code to see if it worked.
if SQLCA.sqlcode = 0 then

// Since we got a row, fetch it and display it.
    FETCH get_emp_proc INTO :emp_name_var;

// Display the employee name.
    MessageBox("Got my employee!",emp_name_var)

// You are all done, so close the procedure.
    CLOSE Get_emp_proc;
```

```
end if
```

PowerBuilder also provides access to return values and output parameters. The return values and output parameters are always in the last result set returned by the stored procedure and they are in this order:

```
return value, output parm1, output parm 2 ...
```

Example 3

Assume the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed. Also assume the description of the SQL Server procedure return is:

```
CREATE PROCEDURE emp_return @m1 int, @m2 int,
@resultp int output
AS SELECT @RESULTP = @m1*@m2
RETURN 0
```

where @m1, @m2, and @resultp are integers.

This example shows how PowerBuilder provides access to return values:

```
//Stored procedure syntax
CREATE PROCEDURE sp_outputs @ml int, @m2 int,
@result int output as SELECT
@result = @ml*@m2;

//Declare syntax in script.
DECLARE myproc PROCEDURE for sp_outputs @ml = 3,
@m2 = 3, @result = 0 output;

//Note: The parameters in the declare must match
//exactly the parameters in the sp.
EXECUTE myproc;

//Execute fetches needed until rc = 100
//then fetch output parameters.
int myresult

FETCH myproc into :myresult;
CLOSE myproc;
```

Microsoft SQL Server Temporary tables

Database stored procedures frequently contain temporary tables that are used as repositories when accumulating rows during processing within the procedure. Since versions of SQL Server prior to SQL Server 2000 do not allow Data Definition Language (DDL) to be executed within the scope of a transaction, PowerBuilder provides the boolean AutoCommit property in the transaction object to allow you to handle these cases.

When AutoCommit is false (the default), normal transaction processing takes place: a BEGIN TRANSACTION is internally issued on a successful connect and this transaction is terminated by a COMMIT TRANSACTION or ROLLBACK TRANSACTION.

When AutoCommit is set to true, no transaction management is performed. Therefore, stored procedures that create temporary tables can be executed. This option should be used with great care because of the recovery implications. If AutoCommit is true, ROLLBACK *cannot* be issued.

See also

Microsoft SQL Server Using CONNECT, COMMIT, DISCONNECT, and ROLLBACK
Microsoft SQL Server Performance and locking

Microsoft SQL Server Using database stored procedures in DataWindow objects

You can use database stored procedures as a data source for DataWindow objects. The following rules apply:

- **Result set definition** You must define what the result set looks like. The DataWindow object cannot determine this information from the stored procedure definition in the database.
- **DataWindow updates** You cannot perform DataWindow updates through stored procedures (that is, you cannot update the database with changes made in the DataWindow object); only retrieval is allowed. (However, the DataWindow can have update characteristics set manually through the DataWindow painter.)
- **Result set processing** You can specify only one result set to be processed when you define the stored procedure result set in the DataWindow painter.

- **Computed rows** Computed rows cannot be processed in a DataWindow.

Microsoft SQL Server Database stored procedures summary

When you use database stored procedures in a PowerBuilder application, keep the following points in mind:

- **Manipulating stored procedures** To manipulate database stored procedures, PowerBuilder provides SQL statements that are similar to cursor statements.
- **Retrieval and update** PowerBuilder supports retrieval, update, or a combination of retrieval and update in database stored procedures, including procedures that return no results sets and those that return one or more result sets.
- **Transactions and stored procedures with result sets** When a procedure executes successfully using a specific connection (transaction) and returns at least one result set, no other SQL commands can be executed using that connection until the procedure has been closed.
- **Transactions and stored procedures without result sets** When a procedure executes successfully using a specific transaction but does not return a result set, the procedure is no longer active. No result sets are pending, and therefore a CLOSE statement is not required.

CHAPTER 23 Using Embedded SQL with Oracle

About this chapter

When you create scripts for a PowerBuilder application, you can use embedded SQL statements in the script to perform operations on the database. The features supported when you use embedded SQL depend on the DBMS to which your application connects.

Overview

When your PowerBuilder application connects to an Oracle database, you can use embedded SQL in your scripts.

If you are using these interfaces to connect to an Oracle database, you can embed the following types of SQL statements in scripts and user-defined functions:

- Transaction management statements
- Non-cursor statements
- Cursor statements
- Database stored procedures

When you use Oracle database interfaces, PowerBuilder supports SQL CREATE TYPE and CREATE TABLE statements for Oracle user-defined types (objects) in the ISQL view of the Database painter. It correctly handles SQL SELECT, INSERT, UPDATE, and DELETE statements for user-defined types in the Database and DataWindow painters.

Oracle Call Interface (OCI)

The Oracle database interfaces use the Oracle Call Interface (OCI) to interact with the database.

When you use embedded SQL, PowerBuilder makes the required calls to the OCI. Therefore, you do not need to know anything about the OCI to use embedded SQL in PowerBuilder.

See also

Chapter 11, Using Oracle
Oracle SQL functions
Oracle Transaction management statements
Oracle Non-cursor statements
Oracle Cursor statements
Oracle Database stored procedures

Oracle Name qualification

Since PowerBuilder does not inspect all SQL statement syntax, you can qualify Oracle catalog entities as necessary.

For example, all of the following qualifications are acceptable:

- emp_name
- employee.emp_name
- jpl.employee.emp_name

Oracle SQL functions

In SQL statements, you can use any function that Oracle supports (such as aggregate or mathematical functions).

For example, you can use the Oracle function UPPER in a SELECT statement:

```
SELECT UPPER(emp_name)
    INTO :emp_name_var
    FROM employee;
```

Calling OCI functions

While PowerBuilder provides access to a large percentage of the features within Oracle, in some cases you may want to call one or more OCI functions directly. In PowerBuilder you can use external function declarations to access any Windows DLL.

The OCI calls qualify for this type of access. Most OCI calls require a pointer to an LDA_DEF structure as their first parameter. If you want to call OCI functions without reconnecting to the database to get an LDA_DEF pointer, use the PowerScript DBHandle function.

DBHandle

DBHandle takes a transaction object as a parameter and returns a long variable, which is the handle to the database for the transaction. This handle is actually the LDA_DEF pointer that PowerBuilder uses internally to communicate with the database. You can use the returned value in your DLLs and pass it as one of the parameters in your function.

Example

This example shows how to use DBHandle. Assume a successful connection has occurred using the default transaction object (SQLCA):

```
// Define a variable to hold our DB handle.
long    OracleHandle

// Get the handle.
OracleHandle = SQLCA.DBHandle( )

// Now that you have the LDA_DEF pointer,
// call the DLL function.
MyDLLFunction( OracleHandle, parm1, parm2, ... )
```

In your DLL, cast the incoming long value into a pointer to an ORA_CSA:

```
VOID FAR PASCAL MyDLLFunction( long lOracleHandle,
    parm1_type parm1,
    parm2_type parm2, ... )
{

// pLda will provide addressability to the Oracle
// logon data area
Lda_Def FAR *pLda = (Lda_Def FAR *)lOracleHandle;

// pCda will point to an Oracle cursor
Cda_Def FAR *pCda = &
        GlobalAllocPtr(GMEM_MOVEABLE,sizeof(Cda_Def));
if(! pCda )

// handle error...
if(open(pCda, pLda,NULL, -1, -1, NULL, -1))

// handle error...
#ifdef Oracle7

// parse the DELETE statement
if(osql3(pCda,
    "DELETE FROM EMPLOYEE WHERE Emp_ID = 100", -1);

#else
if(oparse(pCda,
    "DELETE FROM EMPLOYEE
        WHERE Emp_ID = 100", -1, 0, 1) :
#endif

// handle error...
    if(oclose(pCda))

// handle error...
    GlobalFreePtr(pCda);
}
```

Oracle Transaction management statements

You can use the following transaction management statements with one or more transaction objects to manage connections and transactions for an Oracle database:

- CONNECT
- DISCONNECT
- COMMIT
- ROLLBACK

See also

Oracle Using CONNECT, DISCONNECT, COMMIT, and ROLLBACK

Oracle Using CONNECT, DISCONNECT, COMMIT, and ROLLBACK

The following table lists each transaction management statement and describes how it works when you use any Oracle interface to connect to a database:

Statement	Description
CONNECT	Establishes the database connection. After you assign values to the required properties of the transaction object, you can execute a CONNECT. After the CONNECT completes successfully, PowerBuilder automatically starts an Oracle transaction. This is the start of a logical unit of work.
DISCONNECT	Terminates a successful connection. DISCONNECT automatically executes a COMMIT to guarantee that all changes made to the database since the beginning of the current unit of work are committed.
COMMIT	COMMIT terminates the logical unit of work, guarantees that all changes made to the database since the beginning of the current unit of work become permanent, and starts a new logical unit of work.
ROLLBACK	ROLLBACK terminates a logical unit of work, undoes all changes made to the database since the beginning of the logical unit of work, and starts a new logical unit of work.

Note Oracle does not support the AutoCommit property of the transaction object.

See also

Oracle Performance and locking

Oracle Performance and locking

An important consideration when designing a database application is deciding when CONNECT and COMMIT statements should occur to maximize performance and limit locking and resource use. A CONNECT takes a certain amount of time and can tie up resources during the life of the connection. If this time is significant, then limiting the number of CONNECTs is desirable.

After a connection is established, SQL statements can cause locks to be placed on database entities. The more locks there are in place at a given moment in time, the more likely it is that the locks will hold up another transaction.

Rules

No set of rules for designing a database application is totally comprehensive. However, when you design a PowerBuilder application, you should do the following:

- **Long-running connections** Determine whether you can afford to have long-running connections. If not, your application should connect to the database only when absolutely necessary. After all the work for that connection is complete, the transaction should be disconnected.

  If long-running connections are acceptable, then COMMITs should be issued as often as possible to guarantee that all changes do in fact occur. More importantly, COMMITs should be issued to release any locks that may have been placed on database entities as a result of the statements executed using the connection.

- **SetTrans or SetTransObject function** Determine whether you want to use default DataWindow transaction processing (the SetTrans function) or control the transaction in a script (the SetTransObject function).

  If you cannot afford to have long-running connections and therefore have many short-lived transactions, use the default DataWindow transaction processing. If you want to keep connections open and issue periodic COMMITs, use the SetTransObject function and control the transaction yourself.

Example 1

This script uses embedded SQL to connect to a database and insert a row in the ORDER_HEADER table and a row in the ORDER_ITEM table. Depending on the success of the statements in the script, the script executes a COMMIT or ROLLBACK.

```
// Set the SQLCA connection properties.
SQLCA.DBMS = "O73"
SQLCA.servername = "@TNS:SHOPFLR"
SQLCA.logid = "JPL"
SQLCA.logpass = "STUMP"

// Connect to the database.
CONNECT USING SQLCA;

// Insert a row into the ORDER_HEADER table.
// A ROLLBACK is required only if the first row
// was inserted successfully.
INSERT INTO ORDER_HEADER (ORDER_ID, CUSTOMER_ID)
    VALUES ( 7891, 129 );

// Test return code for ORDER_HEADER insertion.
If SQLCA.sqlcode = 0 then

// Since the ORDER_HEADER is inserted,
// try to insert ORDER_ITEM.
    INSERT INTO ORDER_ITEM &
            (ORDER_ID,ITEM_NBR,PART_NBR,QTY)
```

```
      VALUES ( 7891, 1, '991PLS', 456 );
// Test return code for ORDER_ITEM insertion.
   If SQLCA.sqlcode = -1 then
// The insert failed.
// Roll back insertion of ORDER_HEADER.
      ROLLBACK USING SQLCA;
   End If
End If
COMMIT USING SQLCA;
// Disconnect from the database.
DISCONNECT USING SQLCA;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

This example uses the scripts for the Open and Close events in a window and the Clicked event in a CommandButton to illustrate how you can manage transactions in a DataWindow control. Assume the window contains a DataWindow control dw_1 and the user enters data in dw_1 and then clicks the Cb_Update button to send the data to the database.

Since this script uses SetTransObject to connect to the database, the programmer is responsible for managing the transaction.

The window Open event script:

```
// Set the transaction object properties
// and connect to the database.
// Set the SQLCA connection properties.
SQLCA.DBMS = "O73"
SQLCA.servername = "@TNS:SHOPFLR"
SQLCA.logid = "JPL"
SQLCA.logpass = "STUMP"
// Connect to the database.
CONNECT USING SQLCA;
// Tell the DataWindow which transaction object
// to use.
dw_1.SetTransObject( SQLCA )
```

The CommandButton Clicked event script:

```
// Declare ReturnValue an integer.
integer    ReturnValue

// Update dw_1.
ReturnValue = dw_1.Update( )

// Test to see whether the updates were successful.
If ReturnValue = -1 then

// The updates were not successful.
// Roll back any changes made to the database.
   ROLLBACK USING SQLCA;
Else

// The updates were successful.
// Commit any changes made to the database.
   COMMIT USING SQLCA;
End If
```

The window Close event script:

```
// Since we used SetTransObject,
// disconnect from the database.
DISCONNECT USING SQLCA;
```

Oracle Non-cursor statements

The statements that do not involve cursors are:

- DELETE (Oracle DELETE, INSERT, and UPDATE)
- INSERT (Oracle DELETE, INSERT, and UPDATE)
- Oracle SELECT (singleton)
- UPDATE (Oracle DELETE, INSERT, and UPDATE)

Oracle DELETE, INSERT, and UPDATE

Internally, PowerBuilder processes DELETE, INSERT, and UPDATE statements the same way. PowerBuilder inspects them for any PowerScript variable references and replaces all references with a constant that conforms to Oracle rules for the data type.

Example

Assume you enter the following statement:

```
DELETE FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var is a PowerScript variable with the data type of integer that has been defined within the scope of the script that contains the DELETE statement. Before the DELETE statement is executed, emp_id_var is assigned a value (say 691) so that when the DELETE statement executes, the database receives the following statement:

```
DELETE FROM employee WHERE emp_id = 691;
```

When is this substitution technique used?

This variable substitution technique is used for all PowerScript variable types. When you use embedded SQL, precede all PowerScript variables with a colon (:).

See also

Oracle SELECT

Oracle SELECT

The SELECT statement contains input variables and output variables.

- Input variables are passed to the database as part of the execution and the substitution as described above for DELETE, INSERT, AND UPDATE.
- Output variables are used to return values based on the result of the SELECT statement.

Example 1

Assume you enter the following statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

In this example, emp_id_var, emp_salary_var, and emp_name_var are variables defined within the scope of the script that contains the SELECT statement, and emp_id_var is processed as described in the DELETE example above.

Both emp_name_var and emp_salary_var are output variables that will be used to return values from the database. The data types of emp_name_var and emp_salary_var should be the PowerScript data types that best match the Oracle data type. When the data types do not match perfectly, PowerBuilder converts them.

How big should numeric output variables be?
For numeric data, the output variable must be large enough to hold any value that may come from the database.

Assume the value for emp_id_var is 691 as in the previous example. When the SELECT statement executes, the database receives the following statement:

```
SELECT emp_name,emp_salary
   FROM employee WHERE emp_id=691;
```

If the statement executes with no errors, data locations for the result fields are bound internally. The data returned into these locations is then converted as necessary, and the appropriate PowerScript variables are set to those values.

Example 2

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. It also assumes the data type of the emp_id column in the employee table is CHARACTER[10].

The user enters an employee ID into the line edit sle_Emp and clicks the button Cb_Delete to delete the employee.

The script for the Clicked event in the CommandButton Cb_Delete is:

```
// Make sure we have a value.
if sle_Emp.text <> "" then

// Since we have a value, let's try to delete it.
   DELETE FROM employee
      WHERE emp_id = :sle_Emp.text;

// Test to see if the DELETE worked.
   if SQLCA.sqlcode = 0 then

// It seems to have worked, let user know.
      MessageBox( "Delete",&
         "The delete processed successfully!")
   else

// It didn't work.
      MessageBox("Error", &
         "The delete failed. Invalid Employee ID")
   end if
else

// No input value. Prompt user.
   MessageBox( "Error", &
   "An employee ID is required for delete!")
end if
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 3

This example assumes the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has executed. The user wants to extract rows from the employee table and insert them into the table named extract_employees. The extraction occurs when the user clicks the button Cb_Extract. The boolean variable YoungWorkers is set to TRUE or FALSE elsewhere in the application.

The script for the Clicked event for the CommandButton Cb_Extract is:

```
integer    EmployeeAgeLowerLimit
integer    EmployeeAgeUpperLimit

// Do they have young workers?
if (YoungWorkers = TRUE ) then

// Yes - set the age limit in the YOUNG range.
// Assume no employee is under legal working age.
    EmployeeAgeLowerLimit = 16

// Pick an upper limit.
    EmployeeAgeUpperLimit = 42
else

// No - set the age limit in the OLDER range.
    EmployeeAgeLowerLimit = 43

// Pick an upper limit that includes all
// employees.
    EmployeeAgeUpperLimit = 200
end if

INSERT INTO extract_employee (emp_id,emp_name)
    SELECT emp_id, emp_name FROM employee
       WHERE emp_age >= :EmployeeAgeLowerLimit AND
       emp_age <= :EmployeeAgeUpperLimit;
```

Oracle Cursor statements

In embedded SQL, statements that retrieve data and statements that update data can both involve cursors.

Retrieval statements

The retrieval statements that involve cursors are:

- DECLARE *cursor_name* CURSOR FOR ...
- OPEN *cursor_name*
- FETCH *cursor_name* INTO ...
- CLOSE *cursor_name*

Update statements

The update statements that involve cursors are:

- UPDATE ... WHERE CURRENT OF *cursor_name*
- DELETE ... WHERE CURRENT OF *cursor_name*

PowerBuilder supports all Oracle cursor features.

See also

Oracle Cursor support summary
Oracle Retrieval
Oracle Update

Oracle Retrieval

Retrieval using cursors is conceptually similar to retrieval in the singleton SELECT. The main difference is that since there can be multiple rows in a result set, you control when the next row is fetched into the PowerScript variables.

If you expect only a single row to exist in the employee table with the specified emp_id, use the singleton SELECT. In a singleton SELECT, you specify the SELECT statement and destination variables in one concise SQL statement:

```
SELECT emp_name, emp_salary
    INTO :emp_name_var, :emp_salary_var
    FROM employee WHERE emp_id = :emp_id_var;
```

However, if the SELECT may return multiple rows, you must:

1 Declare a cursor.

2 Open it (which conceptually executes the SELECT).

3 Fetch rows as needed.

4 Close the cursor.

Declaring and opening a cursor

Declaring a cursor is tightly coupled with the OPEN statement. The DECLARE specifies the SELECT statement to be executed, and the OPEN actually executes it.

Declaring a cursor is similar to declaring a variable; a cursor is a nonexecutable statement just like a variable declaration. The first step in declaring a cursor is to define how the result set looks. To do this, you need a SELECT statement, and since you must refer to the result set in subsequent SQL statements, you must associate the result set with a logical name.

Note For UPDATE ... WHERE CURRENT OF *cursor_name* and DELETE ... WHERE CURRENT OF *cursor_name* statements to execute successfully, the SELECT statement must contain the FOR UPDATE clause.

Example

Assume the SingleLineEdit sle_1 contains the state code for the retrieval:

```
// Declare cursor emp_curs for employee table
// retrieval.
DECLARE emp_curs CURSOR FOR
    SELECT emp_id, emp_name FROM EMPLOYEE
    WHERE emp_state = :sle_1.text;

// For UPDATE WHERE CURRENT OF cursor_name and
// DELETE WHERE CURRENT OF cursor_name to work
// correctly in Oracle 7, include the FOR UPDATE
// clause in the SELECT statement.

// Declare local variables for retrieval.
string    emp_id_var
string    emp_name_var

// Execute the SELECT statement with
// the current value of sle_1.text.
OPEN emp_curs;

// At this point, if there are no errors,
// the cursor is available for further processing.
```

Fetching Rows

In the singleton SELECT, you specify variables to hold the values for the columns within the selected row. The FETCH statement syntax is similar to the syntax of the singleton SELECT. Values are returned INTO a specified list of variables.

This example continues the previous example by retrieving some data:

```
// Get the first row from the result set.
FETCH emp_curs INTO :emp_id_var, :emp_name_var;
```

If at least one row can be retrieved, this FETCH places the values of the `emp_id` and `emp_name` columns from the first row in the result set into the PowerScript variables `emp_id_var` and `emp_name_var`. FETCH statements typically occur in a loop that processes several rows from a result set (one row at a time), but that is not the only way they are used.

What happens when the result set is exhausted?
FETCH returns +100 (not found) in the SQLCode property within the referenced transaction object. This is an informational return code; -1 in SQLCode indicates an error.

Closing the cursor

The CLOSE statement terminates processing for the specified cursor. CLOSE releases resources associated with the cursor, and subsequent references to that cursor are allowed only if another OPEN is executed. Although you can have multiple cursors open at the same time, you should close the cursors as soon as possible for efficiency reasons.

Oracle Update

After a FETCH statement completes successfully, you are positioned on a current row within the cursor. At this point, you can execute an UPDATE or DELETE statement using the WHERE CURRENT OF *cursor_name* syntax to update or delete the row. PowerBuilder enforces Oracle cursor update restrictions, and any violation results in an execution error.

Example 1

This cursor example illustrates how you can loop through a result set. Assume the default transaction object (SQLCA) has been assigned valid values and a successful CONNECT has been executed.

The statements retrieve rows from the employee table and then display a message box with the employee name in each row that is found.

```
// Declare the emp_curs cursor.
DECLARE emp_curs CURSOR FOR
    SELECT emp_name FROM EMPLOYEE
        WHERE emp_state = :sle_1.text;

// For UPDATE WHERE CURRENT OF cursor_name and
// DELETE WHERE CURRENT OF cursor_name to work
// correctly in Oracle 7, include the FOR UPDATE
// clause in the SELECT statement.

// Declare a destination variable for employee
// names.
string    emp_name_var

// Execute the SELECT statement with the
// current value of sle_1.text.
OPEN emp_curs;

// Fetch the first row from the result set.
FETCH emp_curs INTO :emp_name_var;

// Loop through result set until exhausted.
DO WHILE SQLCA.sqlcode = 0

// Display a message box with the employee name.
   MessageBox("Found an employee!",emp_name_var)

// Fetch the next row from the result set.
   FETCH emp_curs INTO :emp_name_var;
LOOP

// All done, so close the cursor.
CLOSE emp_curs;
```

Error checking
Although you should test the SQLCode after every SQL statement, these examples show statements to test the SQLCode only to illustrate a specific point.

Example 2

This cursor example illustrates how to use a cursor to update or delete rows. The statements use emp_curs to retrieve rows from the employee table and then ask whether the user wants to delete the employee:

```
// Declare the emp_curs cursor.
DECLARE emp_curs CURSOR FOR
   SELECT  emp_name FROM employee
      WHERE emp_state = :sle_1.text;

// Declare a destination variable for employee
// names.
string emp_name_var

// Declare a return variable for the MessageBox.
int    return_var

// Execute the SELECT statement with the current
// value of sle_1.text.
```

```
OPEN emp_curs;

// Fetch the first row from the result set.
FETCH emp_curs INTO :emp_name_var;

// Loop through result set until it is
// exhausted.
DO WHILE SQLCA.sqlcode = 0

// Ask the user to confirm the deletion.
    return_var = MessageBox( "Want to delete?",&
    emp_var_name, Question!, YesNo!, 2 )

// Delete?
If ( return_var = 1 ) then

// Yes - delete the employee.
          DELETE FROM employee
              WHERE CURRENT OF emp_curs;
       End If

// Fetch the next row from the result set.
    FETCH emp_curs INTO :emp_name_var;
LOOP

// All done, so close the cursor.
CLOSE emp_curs;
```

Oracle Cursor support summary

When you use cursors with any Oracle interface, keep the following points in mind:

- Oracle provides native support for cursors.
- PowerBuilder supports retrieval using cursors.
- PowerBuilder supports delete or update using cursors.

Oracle Database stored procedures

Oracle stored procedures

If your database is Oracle Version 7.2 or higher, you can use an Oracle stored procedure that has a result set as an IN OUT (reference) parameter.

Procedures with a single result set You can use stored procedures that return a single result set in DataWindow objects, reports, and embedded SQL, but not when using the RPCFUNC keyword to declare the stored procedure as an external function or subroutine.

Procedures with multiple result sets You can use stored procedures that return multiple result sets only in embedded SQL. Multiple result sets are not supported in DataWindow objects, reports, or with the RPCFUNC keyword.

The O90 database interface supports SQL CREATE TYPE and CREATE TABLE statements for Oracle user-defined types (objects) in the ISQL view of the Database painter. It correctly handles SQL SELECT, INSERT, UPDATE, and DELETE statements for user-defined types in the Database and DataWindow painters. For more information, see Chapter 11, Using Oracle.

Methods for using Oracle stored procedures

There are three methods for using Oracle stored procedures in a PowerBuilder application:

- **As a data source** for DataWindow objects.
- **RPCFUNC keyword (Recommended)** Use the RPCFUNC keyword to declare the stored procedure as an external function or external subroutine. You cannot use the RPCFUNC keyword with Oracle stored procedures that return result sets. Using the RPCFUNC keyword to declare the stored procedure provides the best performance and has more supported features and fewer limitations than the DECLARE Procedure and PBDBMS methods.
- **DECLARE Procedure statement** Use the DECLARE Procedure (Oracle DECLARE and EXECUTE) statement to declare the stored procedure as an external function or external subroutine. This includes support for fetching against Oracle stored procedures that return result sets.

See also

Supported features when using Oracle stored procedures
Using DECLARE, EXECUTE, FETCH, and CLOSE with Oracle stored procedures

Supported features when using Oracle stored procedures

Supported features with RPCFUNC keyword

The following are supported and unsupported Oracle PL/SQL features when you use the RPCFUNC keyword to declare the stored procedure:

You can	You cannot
Use IN, OUT, and IN OUT parameters	Pass and return records
Use an unlimited number of parameters	
Overload procedures	
Pass and return PowerScript arrays (PL/SQL tables)	
Use function return codes	
Use blobs up to 32,512 bytes long as parameters	

Supported features with DECLARE Procedure statement

The following are supported and unsupported Oracle PL/SQL features when you use the DECLARE Procedure statement:

You can	You cannot
Use IN and OUT parameters	Use IN OUT parameters
Use up to 256 parameters	Pass and return records
	Use more than 256 parameters
	Pass and return PowerScript arrays (PL/SQL tables)
	Overload procedures

For an example that uses a REF CURSOR variable of type IN OUT, see Chapter 11, Using Oracle.

Using DECLARE, EXECUTE, FETCH, and CLOSE with Oracle stored procedures

PowerBuilder provides SQL statements that are very similar to cursor operations to support retrieval using database stored procedures. In PowerBuilder embedded SQL, there are four commands that involve database stored procedures:

- DECLARE *procedure_name* PROCEDURE FOR ... (Oracle DECLARE and EXECUTE)
- EXECUTE *procedure_name* (Oracle DECLARE and EXECUTE)
- FETCH *procedure_name* INTO ... (Oracle FETCH)
- CLOSE *procedure_name* (Oracle CLOSE)

Oracle DECLARE and EXECUTE

PowerBuilder requires a declarative statement to identify the database stored procedure that is being used and a logical name that can be referenced in subsequent SQL statements. The general syntax for declaring a procedure is:

```
DECLARE logical_procedure_name PROCEDURE FOR
   Oracle_procedure_name(:InParam1,:InParam2, ...)
   {USING transaction_object};
```

where *logical_procedure_name* can be any valid PowerScript data identifier and *Oracle_procedure_name* is the name of the stored procedure in the database.

The parameter references can take the form of any valid parameter string that Oracle accepts. PowerBuilder does not inspect the parameter list format except for purposes of variable substitution. The USING clause is required only if you are using a transaction object other than the default transaction object.

You can use Oracle Named or Positional notation to specify the procedure arguments. Positional is simpler to specify, but you must use Named if any output parameters are defined to the left of any input parameters.

Example 1

If a stored procedure is defined as:

```
CREATE PROCEDURE spm1
   (dept varchar2, mgr_name OUT varchar2)
   IS lutype varchar2(10);
   BEGIN
   SELECT manager INTO mgr_name FROM mgr_table
   WHERE dept_name = dept;
   END;
```

To declare that procedure for processing within PowerBuilder, you code:

```
DECLARE dept_proc PROCEDURE FOR
   spm1(:dept);
```

Note that this declaration is a non-executable statement, just like a cursor declaration. Where cursors have an OPEN statement, procedures have an EXECUTE statement.

When the EXECUTE statement executes, the procedure is invoked. The EXECUTE refers to the logical procedure name.

```
EXECUTE dept_proc;
```

Example 2

The following example that declares a function in a service object that reads a pipe shows the use of named notation:

```
public function integer f_GetId (string as_PipeName)
double ldbl_Id

DECLARE f_GetId PROCEDURE FOR
    f_GetId (pipe_name => :as_PipeName) USING SQLCA;

EXECUTE f_GetId;

FETCH f_GetId INTO :ldbl_Id;

CLOSE f_GetId;

RETURN ldbl_Id;
```

Example 3

Given this procedure:

```
CREATE OR REPLACE PROCEDURE spu_edt_object(
o_id_object OUT NUMBER,
o_message OUT VARCHAR2,
a_id_object NUMBER,
a_param VARCHAR2 := NULL,
a_value VARCHAR2 := NULL
) as
begin
o_id_object := 12345;
o_message := 'Hello World';
end;
```

The DECLARE statement must use named notation because output parameters are defined to the left of input parameters:

```
dec{0} o_id_object, id_obiect = 54321
string o_message, param = 'Test'

DECLARE proc_update PROCEDURE FOR spu_edt_object (
a_id_object => :id_object,
a_param => :param
)
USING SQLCA;
```

```
EXECUTE proc_update;
if SQLCA.SqlCode 0 then
SQLCA.f_out_error()
RETURN -1
end if

FETCH proc_update INTO :o_id_object, o_message;
if SQLCA.SqlCode 0 then
SQLCA.f_out_error()

RETURN -1
end if
```

Oracle FETCH

To access rows returned by a procedure, you use the FETCH statement as you did for cursors. You can execute the FETCH statement after any EXECUTE statement that executes a procedure that has output parameters.

Example

```
FETCH dept_proc INTO :name_var;
```

The FETCH FROM *procedure* statements must exactly match the output parameters returned by the stored procedure when it executes.

Oracle CLOSE

If a database stored procedure has output parameters, it must be closed when processing is complete.

Closing a procedure looks the same as closing a cursor.

Example

```
CLOSE dept_proc;
```

PART 7

Appendix

The Appendix describes how to modify the PBODB170 initialization file.

APPENDIX

Adding Functions to the PBODB170 Initialization File

About this appendix

Usually, *you do not need to modify the PBODB170 initialization file*. In certain situations, however, you might need to add functions to the PBODB170 initialization file for connections to your back-end DBMS through the ODBC or OLE DB interface in PowerBuilder.

This appendix describes how to add functions to the PBODB170 initialization file if necessary.

Contents

About the PBODB170 initialization file

What is the PBODB170 initialization file?

When you access data through the ODBC interface, PowerBuilder uses the PBODB170 initialization file (*PBODB170.INI*) to maintain access to extended functionality in the back-end DBMS for which ODBC does not provide an API call. Examples of extended functionality are SQL syntax or function calls specific to a particular DBMS.

Editing PBODB170.INI

In most cases, you do *not* need to modify *PBODB170.INI*. Changes to this file can adversely affect PowerBuilder. Change *PBODB170.INI* only if you are asked to do so by a Technical Support representative.

However, you *can* edit *PBODB170.INI* if you need to add functions for your back-end DBMS.

If you modify *PBODB170.INI*, first make a copy of the existing file. Then keep a record of all changes you make. If you call Technical Support after modifying *PBODB170.INI*, tell the representative that you changed the file and describe the changes you made.

Adding functions to PBODB170.INI

PBODB170.INI lists the functions for certain DBMSs that have ODBC drivers. If you need to add a function to *PBODB170.INI* for use with your back-end DBMS, you can do either of the following:

- **Existing sections** Add the function to the Functions section for your back-end database if this section exists in *PBODB170.INI*.
- **New sections** Create new sections for your back-end DBMS in *PBODB170.INI* and add the function to the newly created Functions section.

Adding functions to an existing section in the file

If sections for your back-end DBMS *already exist* in *PBODB170.INI*, use the following procedure to add new functions.

❖ To add functions to an existing section in PBODB170.INI:

1 Open *PBODB170.INI* in one of the following ways:

- Use the File Editor in PowerBuilder. (For instructions, see the *Users Guide*.)
- Use any text editor outside PowerBuilder.

2 Locate the entry for your back-end DBMS in the DBMS Driver/DBMS Settings section of *PBODB170.INI*.

For example, here is the*PBODB170.INI* entry for SQL Anywhere:

```
;*********************************************
;DBMS Driver/DBMS Settings see comments at end
;of file
;*********************************************
...
[SQL Anywhere]
PBSyntax='WATCOM50_SYNTAX'
PBDateTime='STANDARD_DATETIME'
PBFunctions='ASA_FUNCTIONS'

PBDefaultValues='autoincrement,current date,
   current time,current timestamp,timestamp,
   null,user'
PBDefaultCreate='YES'
PBDefaultAlter='YES'
```

```
PBDefaultExpressions='YES'
DelimitIdentifier='YES'
PBDateTimeInvalidInSearch='NO'
PBTimeInvalidInSearch='YES'
PBQualifierIsOwner='NO'

PBSpecialDataTypes='WATCOM_SPECIALDATATYPES'
IdentifierQuoteChar='"'
PBSystemOwner='sys,dbo'
PBUseProcOwner='YES'
SQLSrvrTSName='YES'
SQLSrvrTSQuote='YES'
SQLSrvrTSDelimit='YES'
ForeignKeyDeleteRule='Disallow if Dependent Rows
   Exist (RESTRICT),Delete any Dependent Rows
   (CASCADE),Set Dependent Columns to NULL
   (SET NULL)'
TableListType='GLOBAL TEMPORARY'
```

3 Find the name of the section in *PBODB170.INI* that contains function information for your back-end DBMS.

To find this section, look for a line similar to the following in the DBMS Driver/DBMS Settings entry:

```
PBFunctions='section_name'
```

For example, the following line in the DBMS Driver/DBMS Settings entry for SQL Anywhere indicates that the name of the Functions section is ASA_FUNCTIONS:

```
PBFunctions='ASA_FUNCTIONS'
```

4 Find the Functions section for your back-end DBMS in *PBODB170.INI*.

For example, here is the Functions section for SQL Anywhere:

```
;*********************************************
;Functions
;*********************************************
[ASA_FUNCTIONS]
AggrFuncs=avg(x),avg(distinct x),count(x),
   count(distinct x),count(*),list(x),
   list(distinct x),max(x),max(distinct x),
   min(x),min(distinct x),sum(x),sum(distinct x)

Functions=abs(x),acos(x),asin(x),atan(x),
   atan2(x,y),ceiling(x),cos(x),cot(x),degrees(x),
   exp(x),floor(x),log(x),log10(x),
   mod(dividend,divisor),pi(*),power(x,y),
   radians(x),rand(),rand(x),
   remainder(dividend,divisor),round(x,y),
   sign(x),sin(x),sqrt(x),tan(x),
   "truncate"(x,y),ascii(x),byte_length(x),
   byte_substr(x,y,z),char(x),char_length(x),
   charindex(x,y),difference(x,y)insertstr(x,y,z),

   lcase(x),left(x,y),length(x), locate(x,y,z),
   lower(x),ltrim(x),patindex('x',y),repeat(x,y),
   replicate(x,y),right(x,y),rtrim(x),
   similar(x,y),soundex(x),space(x),str(x,y,z),
   string(x,...),stuff(w,x,y,z),substr(x,y,z),
   trim(x),ucase(x),upper(x),date(x),
   dateformat(x,y),datename(x,y),day(x),
   dayname(x),days(x),dow(x),hour(x),hours(x),
   minute(x),minutes(x),minutes(x,y),month(x),
   monthname(x),months(x),months(x,y),now(*),
   quarter(x),second(x),seconds(x),seconds(x,y),
   today(*),weeks(x),weeks(x,y),year(x),years(x),
   years(x,y),ymd(x,y,z),dateadd(x,y,z),
   datediff(x,y,z),datename(x,y),datepart(x,y),
   getdate(),cast(x as y),convert(x,y,z),

   hextoint(x),inttohex(x),
   connection_property(x,...),datalength(x),
   db_id(x),db_name(x),db_property(x),
   next_connection(x),next_database(x),
   property(x),property_name(x),
   property_number(x),property_description(x),
   argn(x,y,...),coalesce(x,...),
   estimate(x,y,z),estimate_source(x,y,z),
   experience_estimate(x,y,z),ifnull(x,y,z),
   index_estimate(x,y,z),isnull(x,...),
   number(*),plan(x),traceback(*)
```

5 To add a new function, type a comma followed by the function name at the end of the appropriate function list, as follows:

- **Aggregate functions** Add aggregate functions to the end of the AggrFuncs list.
- **All other functions** Add all other functions to the end of the Functions list.

Case sensitivity
If the back-end DBMS you are using is case sensitive, be sure to use the required case when you add the function name.

The following example shows a new function for SQL Anywhere added at the end of the Functions list:

```
;*********************************************
;Functions
;*********************************************
[ASA_FUNCTIONS]
AggrFuncs=avg(x),avg(distinct x),count(x),
   count(distinct x),count(*),list(x),
   list(distinct x),max(x),max(distinct x),
   min(x),min(distinct x),sum(x),sum(distinct x)
Functions=abs(x),acos(x),asin(x),atan(x),
   atan2(x,y),ceiling(x),cos(x),cot(x),degrees(x),
   exp(x),floor(x),log(x),log10(x),
   mod(dividend,divisor),pi(*),power(x,y),
   radians(x),rand(),rand(x),
   ...
   number(*),plan(x),traceback(*),newfunction()
```

6 Save your changes to *PBODB170.INI*.

Adding functions to a new section in the file

If entries for your back-end DBMS *do not exist* in *PBODB170.INI*, use the following procedure to create the required sections and add the appropriate functions.

Before you start
For more about the settings to supply for your back-end DBMS in *PBODB170.INI*, read the comments at the end of the file.

❖ **To add functions to a new section in PBODB170.INI:**

1 Open *PBODB170.INI* in one of the following ways:

- Use the File Editor in PowerBuilder. (For instructions, see the *Users Guide*.)
- Use any text editor outside PowerBuilder.

2 Edit the DBMS Driver/DBMS Settings section of the PBODB170 initialization file to add an entry for your back-end DBMS.

Finding the name
The name required to identify the entry for your back-end DBMS in the DBMS Driver/DBMS Settings section is in *PBODB170.INI*.

Make sure that you:

- Follow the instructions in the comments at the end of *PBODB170.INI*.
- Use the same syntax as existing entries in the DBMS Driver/DBMS Settings section of *PBODB170.INI*.
- Include a section name for PBFunctions.

For example, here is the relevant portion of an entry for a DB2/2 database:

```
;***********************************************
;DBMS Driver/DBMS Settings
;***********************************************
[DB2/2]
...
PBFunctions='DB22_FUNCTIONS'
...
```

3 Edit the Functions section of *PBODB170.INI* to add an entry for your back-end DBMS.

Make sure that you:

- Follow the instructions in the comments at the end of *PBODB170.INI*.
- Use the same syntax as existing entries in the Functions section of *PBODB170.INI*.
- Give the Functions section the name that you specified for PBFunctions in the DBMS Driver/DBMS Settings entry.

For example:

```
;***********************************************
;Functions
;***********************************************
[DB22_FUNCTIONS]
AggrFuncs=avg(),count(),list(),max(),min(),sum()
Functions=curdate(),curtime(),hour(), ...
```

4 Type a comma followed by the function name at the end of the appropriate function list, as follows:

- **Aggregate functions** Add aggregate functions to the end of the AggrFuncs list.

- **All other functions** Add all other functions to the end of the Functions list.

Case sensitivity
If the back-end DBMS you are using is case sensitive, be sure to use the required case when you add the function name.

The following example shows (in bold) a new DB2/2 function named substr() added at the end of the Functions list:

```
;*********************************************
;Functions
;*********************************************
[DB22_FUNCTIONS]
AggrFuncs=avg(),count(),list(),max(),min(),sum()
Functions=curdate(),curtime(),hour(), substr()
```

5 Save your changes to *PBODB170.INI*.

Index