Accessing Databases

Release 3.5 of Forte™ 4GL

Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 U.S.A. 1-800-786-7638

Part No. 806-6662-01
October 2000, Revision A

Contents

2 Defining a Resource Manager

3 Making a Database Connection

4 Working with Data Types

5 Manipulating Data

Preface

Accessing Databases provides information about accessing databases from Forte applications. This manual is written for two types of readers:

System managers will find instructions for preparing a database management system to work with a Forte application. This primarily entails defining a resource manager for each database that will interact with a Forte program (described in Chapter 2, "Defining a Resource Manager").

Forte *application programmers* will find information they need to write applications using TOOL SQL statements or dynamic SQL methods. We assume that these readers:

- have programming experience
- are familiar with their particular window system
- are familiar with SQL and their particular database management system
- understand the basic concepts of object-oriented programming as described in *A Guide to the Forte 4GL Workshops*
- have used the Forte workshops to create classes

Organization of This Manual

This manual contains a combination of user and reference material about integrating Forte applications with database management systems, and includes the following chapters and appendixes:

Chapter	Description
Chapter 1, "Introduction to the Forte Database Interface"	Provides an overview of Forte support for database integration.
Chapter 2, "Defining a Resource Manager"	Describes how to create a resource manager for database access.
Chapter 3, "Making a Database Connection"	Describes DBSession and DBResourceMgr services objects, how to create each, and their advantages and disadvantages.
Chapter 4, "Working with Data Types"	Discusses the mappings between database data types and Forte TOOL data types, and points out conversion issues.
Chapter 5, "Manipulating Data"	Provides information about writing TOOL SQL statements and SQL methods to access (query) and manipulate (update) database data.
Chapter 6, "Transactions"	Discusses the relationship between database transactions and Forte transactions.
Chapter 7, "Error Handling"	Provides a brief overview of Forte's database exception classes.
Appendix A, "Database Example Applications"	Describes the standard Forte sample applications for database access, with instructions for installation and use.
Appendix B, "TOOL SQL Statement Reference"	Provides the primary source of information about the SQL subset of TOOL statements; for other TOOL statement reference, see the *TOOL Reference Manual*.

Conventions

This manual uses standard Forte documentation conventions in specifying command syntax and in documenting TOOL code.

Command Syntax Conventions

The specifications of command syntax in this manual use a “brackets and braces” format. The following table describes this format:

Format	Description
bold	Bold text is a reserved word; type the word exactly as shown.
italics	Italicized text is a generic term that represents a set of options or values. Substitute an appropriate clause or value where you see italic text.
UPPERCASE	Uppercase text represents a constant. Type uppercase text exactly as shown.
underline	Underlined text represents a default value.
vertical bars |	Vertical bars indicate a mutually exclusive choice between items. See braces and brackets, below.
braces { }	Braces indicate a required clause. When a list of items separated by vertical bars is enclosed in braces, you must enter one of the items from the list. Do not enter the braces or vertical bars.
brackets [ ]	Brackets indicate an optional clause. When a list of items separated by vertical bars is enclosed by brackets, you can either select one item from the list or ignore the entire clause. Do not enter the brackets or vertical bars.
ellipsis …	The item preceding an ellipsis may be repeated one or more times. When a clause in braces is followed by an ellipsis, you can use the clause one or more times. When a clause in brackets is followed by an ellipsis, you can use the clause zero or more times.

TOOL Code Conventions

Where this manual includes documentation or examples of TOOL code, the TOOL code conventions in the following table are used.

Format	Description
parentheses ( )	Parentheses are used in TOOL code to enclose a parameter list. Always include the parentheses with the parameter list.
comma ,	Commas are used in TOOL code to separate items in a parameter list. Always include the commas in the parameter list.
colon :	Colons are used in TOOL code to separate a name from a type, or to indicate a Forte name in a SQL statement. Always include the colon in the type declaration or statement.
semicolon ;	Semicolons are used in TOOL code to end a TOOL statement. Always type a semicolon at the end of a statement.

The Forte Documentation Set

Forte produces a comprehensive documentation set describing the libraries, languages, workshops, and utilities of the Forte Application Environment. The complete Forte Release 3 documentation set consists of the following manuals in addition to comprehensive online Help.

Forte 4GL

- *A Guide to the Forte 4GL Workshops*
- *Accessing Databases*
- *Building International Applications*
- *Escript and System Agent Reference Manual*
- *Forte 4GL Java Interoperability Guide*
- *Forte 4GL Programming Guide*
- *Forte 4GL System Installation Guide*
- *Forte 4GL System Management Guide*
- *Fscript Reference Manual*
- *Getting Started With Forte 4GL*
- *Integrating with External Systems*
- *Programming with System Agents*
- *TOOL Reference Manual*
- *Using Forte 4GL for OS/390*

Forte Express

- *A Guide to Forte Express*
- *Customizing Forte Express Applications*
- *Forte Express Installation Guide*

Forte WebEnterprise and WebEnterprise Designer

- *A Guide to WebEnterprise*
- *Customizing WebEnterprise Designer Applications*
- *Getting Started with WebEnterprise Designer*
- *WebEnterprise Installation Guide*

Forte Example Programs

In this manual, we often include code fragments to illustrate the use of a feature that is being discussed. If a code fragment has been extracted from a Forte example program, the name of the example program is given after the code fragment. If a major topic is illustrated by a Forte example program, reference will be made to the example program in the text.

These Forte example programs come with the Forte product. They are located in subdirectories under $FORTE_ROOT/install/examples. The files containing the examples have a .pex suffix. You can search for TOOL commands or anything of special interest with operating system commands. The .pex files are text files, so it is safe to edit them, though you should only change private copies of the files.

Viewing and Searching PDF Files

You can view and search 4GL PDF files directly from the documentation CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

Note You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from http://www.adobe.com. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

To copy the documentation to a client or server:

1. Copy the fortedoc directory and its contents from the CD-ROM to the client or server hard disk.

   You can specify any convenient location for the fortedoc directory; the location is not dependent on the Forte distribution.

2. Set up a directory structure that keeps the fortedoc.pdf and the 4gl directory in the same relative location.

   The directory structure must be preserved to use the Acrobat search feature.

Note To uninstall the documentation, delete the fortedoc directory.

To view and search the documentation:

1. Open the file fortedoc.pdf, located in the fortedoc directory.
2. Click the **Search** button at the bottom of the page or select **Edit > Search > Query**.
3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click **Search**.

   A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

Note For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

   All occurrences of the word or phrase on a page are highlighted.

5. Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

Toolbar Button	Keyboard Command
Next Highlight	Ctrl+]
Previous Highlight	Ctrl+[
Next Document	Ctrl+Shift+]

6. To return to the fortedoc.pdf file, click the Homepage bookmark at the top of the bookmarks list.
7. To revisit the query results, click the **Results** button at the bottom of the fortedoc.pdf home page or select **Edit > Search > Results**.

Chapter 1

Introduction to the Forte Database Interface

This chapter provides an introduction to the Forte database interface. Database management systems (DBMS) are standard components of many complex applications. The Forte Application Environment includes interfaces to the following database management systems:

- IBM UDB/DB2
- Informix Dynamic Server
- Microsoft SQL Server
- Oracle
- Rdb (both Oracle Rdb and its predecessor, DEC Rdb)
- Sybase

In addition, the Forte database interface also supports Microsoft's Open Database Connectivity (ODBC), allowing access to data sources through database drivers that conform to the ODBC standard.

For information about which database versions or platforms are certified for a given Forte release, you should refer to the platform support matrix, available at http://www.forte.com/support/platforms.html.

About the Forte Database Interface

The Forte database interface allows Forte application developers to build database access into their applications quickly and easily. Application developers can take advantage of virtually all database features, both standard and proprietary, while building applications that are portable across hardware platforms, operating systems, and databases.

The primary Forte tools that support database access are the TOOL SQL statements and Forte's GenericDBMS library. This introduction describes these tools in more detail. Forte also provides a number of sample applications that demonstrate working with database data. These programs are described in Appendix A, "Database Example Applications."

Support for Multiple Database Management Systems

Forte's database interface supports the following database systems:

- IBM UDB/DB2 R6000 & Mainframe
- Informix
- Microsoft SQL Server
- ODBC Level 1 API ODBC 2.1 SDK
- Oracle
- Rdb
- Sybase

For detailed information regarding which versions and platforms of a particular database management system are supported, refer to the *Forte 4GL System Installation Guide*, which contains the following types of information:

- version and release numbers of certified third party components
- notes on platforms that are certified for a given product and release

The *Forte Release Notes* may also contain additional information.

When this manual describes information that is unique to one or more database vendors, the information is identified as *database vendor-specific* information.

Support for Standard and Proprietary SQL

Forte database applications can contain any SQL statement supported by any database vendor, with very few exceptions. An application can contain any combination of DML (data manipulation language, including queries, updates, inserts, and deletes), DDL (data definition language, for creating, altering, and dropping tables), or DCL (data control language, for database access and security). Many applications consist primarily of DML.

ANSI SQL and vendor SQL extensions

You can write portable applications using ANSI standard SQL statements that all vendors support. You can also use vendor-specific SQL extensions, such as outer-joins and advanced set operations, although when you use such extensions, your applications may not be portable across all databases.

The Forte database interface supports SQL access to a relational database system using two different approaches:

- using Forte TOOL SQL statements
- using classes in Forte's GenericDBMS library

Table 1 lists the SQL statements supported by TOOL along with the equivalent method. All methods are on the DBSession class of the GenericDBMS library.

Table 1 *Equivalent TOOL SQL Statements and Methods*

TOOL SQL Statement	DBSession Method	Description
sql select	Select *	Retrieve a single row from a database table or retrieves multiple rows from a database table into an array.
sql delete	Execute *	Remove rows from a database table.
sql insert	Execute *	Add a new row to a database table.
sql update	Execute *	Replace values in a database table.
sql open cursor	OpenCursor *	Open a cursor.
sql close cursor	CloseCursor *	Close a cursor.
sql fetch cursor	FetchCursor *	Retrieve a single row from a cursor or retrieve multiple rows from a cursor into an array.
sql execute immediate	ExecuteImmediate	Execute a single SQL statement specified as a literal string, a string variable, or a TextData variable.
sql execute procedure ** (all databases except Sybase)	OpenCursor (Sybase procedures only)	Execute a database procedure.
* These statements must be previously prepared using the DBSession class interface.		
** This statement must be previously prepared using the PrepareProcedure method of the DBSession class.		

Accessing Databases using TOOL SQL

TOOL SQL statements can be used in a number of situations. If you have a choice of using a TOOL SQL statement or a GenericDBMS class, it is usually simpler to use the TOOL statement.

In general, any operation you can perform using TOOL SQL you can also perform using a GenericDBMS class. For single return values, use the TOOL SQL statement **execute procedure**; for multiple result sets, use the DBSession class.

Appendix B, "TOOL SQL Statement Reference" is the primary reference for TOOL SQL statements. Many of these statements are demonstrated in Chapter 5, "Manipulating Data." For reference information regarding non-SQL TOOL statements, refer to the *TOOL Reference Manual*.

Accessing Databases using the Database Classes

The GenericDBMS library contains a number of classes that support connecting to and working with relational database systems. This library must be named as a supplier plan in order to access any database. You use the Project Workshop to make the GenericDBMS library a supplier plan to the project in which you want to access the database.

TOOL statements are equivalent to GenericDBMS methods

Many GenericDBMS classes and methods, in particular, those on the DBSession class, are functionally equivalent to TOOL SQL statements. For example, when you are writing a method, you can either invoke the Select method on the DBSession class or use the TOOL statement **sql select**; either way provides the same execution, features, and performance.

Other classes in GenericDBMS, such as DBDatabaseDesc, DBTableDesc, DBColumnDesc, and DBKeyDesc are "helper" classes; they do not have TOOL SQL equivalents, but are very useful for returning information about database tables, columns, keys, and so on.

When GenericDBMS methods are required

In some circumstances you must use the classes and methods in GenericDBMS. You must use the classes in the GenericDBMS library to build applications that execute SQL statements that are constructed at runtime. These statements are often used in applications to construct queries or updates based upon input from the end-user. TOOL SQL statements cannot be used to execute SQL statements specified at runtime.

"Using Forte Classes to Execute SQL" on page 85 describes how to use the GenericDBMS classes to build and execute SQL statements at runtime.

The Forte GenericDBMS Library

Every Forte application that interacts with a database system must include the Forte library called GenericDBMS as a supplier plan. This library contains the classes that allow you to use SQL in your applications.

In addition to the GenericDBMS library you also need a vendor-specific library for each database to which you actually connect. For example, if you partition your application to use an Oracle resource manager, then you need the library called Oracle. The database-specific libraries have the name of the database vendor (as in DB2, Informix, and so on).

Always add the GenericDBMS library as a supplier library for any database access, even if you know that you will only use one type of database. Do not explicitly refer to or add the individual database vendor libraries as supplier libraries. They are added dynamically during partitioning, as needed.

The Forte online Help contains reference material for the GenericDBMS library.

How to Access an RDBMS

This manual describes how to integrate a Forte application with a database, so that your Forte application users can query and update database data. To integrate a database with Forte application, you can define a service object whose purpose is to handle calls to the database (you can also build Forte applications without such a service object).

The basic components of a Forte database application configuration are shown in Figure 1.

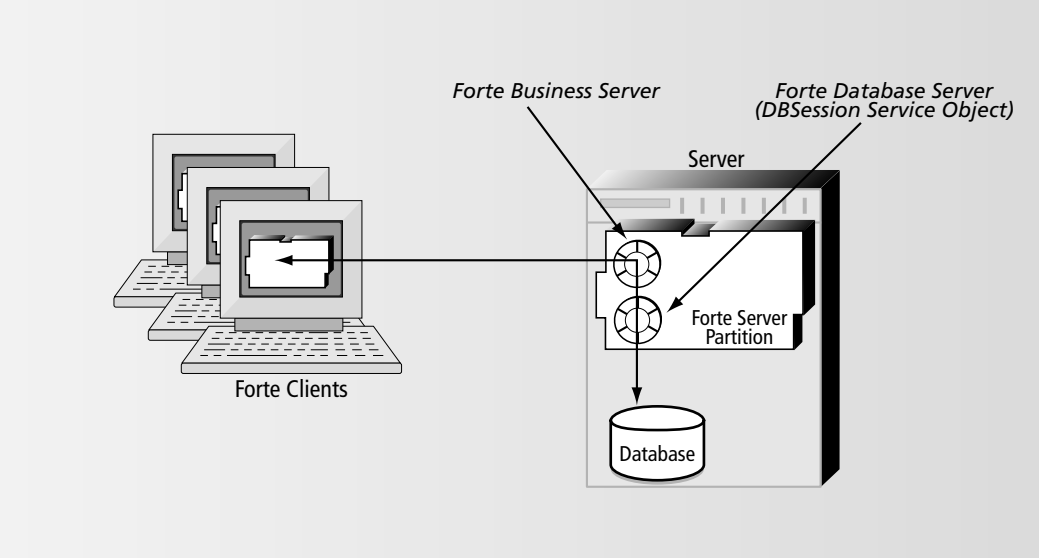


Figure 1 *Components of a Typical Forte Database Application*

Figure 1 does not necessarily represent a typical Forte database application, but rather the minimal components of one. Note that it shows two service objects. Each Forte application service object (there may be several) manages Forte objects and their logic. A DBSession service object (or a DBResourceMgr service object) coordinates communications with the database, and interacts with the Forte service objects. Service objects are sometimes called "servers."

Figure 2 shows a chart of the high-level steps required to build an application such as that shown in Figure 1. This book describes each of these tasks in more detail.

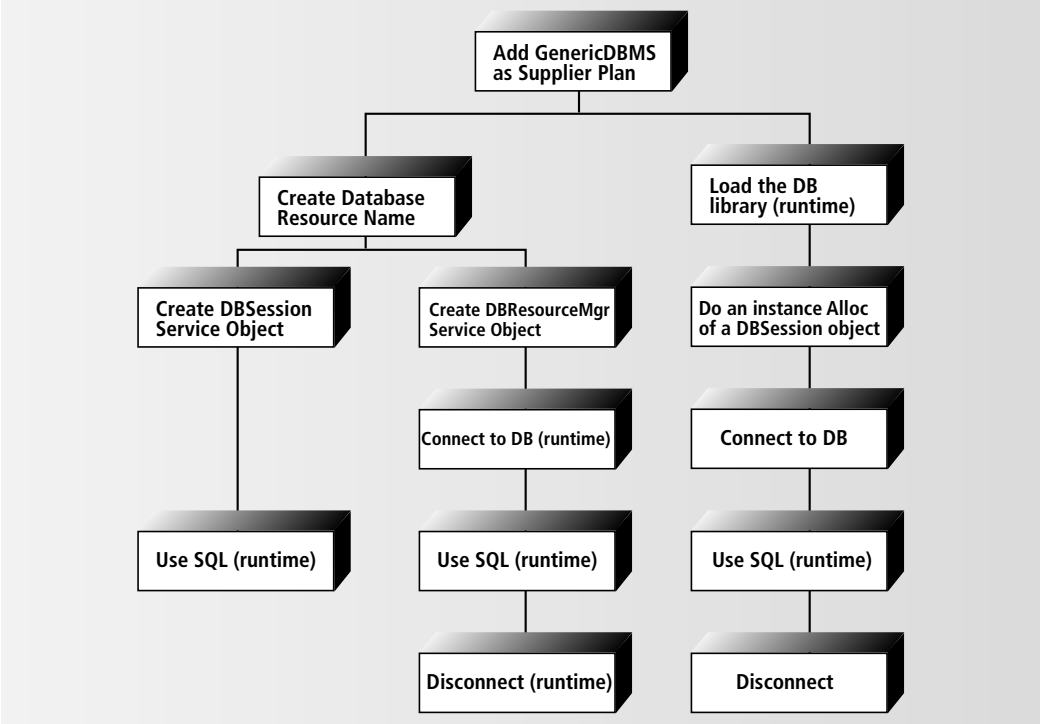


Figure 2 *Steps to Build a Forte Database Application*

The following table lists some of the integration steps, keyed to sections in this manual.

Task	Reference
Include the GenericDBMS library as a supplier plan.	*A Guide to the Forte 4GL Workshops*
Define a database resource name.	"Defining a Resource Name" on page 28
Set database environment variables.	"Environment Variables" on page 29
Create a service object of either type.	"Creating a Database Service Object" on page 40
Use a DBResourceMgr service object.	"When to Use a DBResourceMgr Service Object" on page 39 "Connecting with a DBResourceMgr Service Object" on page 46
Use a DBSession service object.	"When to Use a DBSession Service Object" on page 38 "Connecting with a DBSession Service Object" on page 45
Dynamically select a database to connect to.	"Dynamically Choosing a Database Vendor" on page 47
Run Forte using a standalone database.	"Using a Local Database when Running Forte Standalone" on page 35
Write TOOL SQL statements to query and update database data.	"Using TOOL Statements to Query Data" on page 71
Write methods on DBSession class to generate SQL statements at application runtime.	"Using Forte Classes to Execute SQL" on page 85
Reconnect a dropped database session.	"Reconnecting to a Database Session" on page 52
Disconnect the session.	"Disconnecting a Database Session" on page 52

Database Vendor Notes

Following are some general notes on how the Forte database interface works with specific databases. This type of database vendor-specific information is found throughout the manual; for example, see "Environment Variables" on page 29.

DB2 Notes

In IBM's DB2 Client/Server architecture, a DB2/6000 application executing through the DB2 CLI Driver can connect to the following:

- a local DB2/6000 database using its DB2 server
- a remote DB2/6000 database using a local DB2 Client Application Enabler and a remote DB2 Client Support, using DB2 networking services

When connecting from a Forte application to a DB2/6000 database, the first DB2 configuration above is used. Forte partitions the application to execute the DB2 service object (either a DBResourceMgr or DBSession) on the RS/6000 node where the DB2 database resides. Forte application services are used to communicate between the Forte/DB2 server partition and other application components. The actual DB2 connection, then, is a local connection between the Forte server partition and the DB2/6000 server.

Forte provides array interface support for DB2 and ODBC. This standard can improve network performance when transferring multiple rows between a database and a client. Using the array interface requires only one round trip between the database and client to transfer multiple rows, rather than one round trip per row.

Informix Notes

An Informix database is identified by both an Informix server name and a database name. If a server name is not specified, then the value of the INFORMIXSERVER environment variable is used. Note that, unlike native Informix tools such as *dbaccess*, when Forte connects to Informix a database must be specified; server-only connections are not supported.

Note — A Forte server partition is limited to a single shared memory connection to a local Informix database.

Oracle Notes

A Forte application that integrates with an Oracle RDBMS offers two communication models; both models require SQL*Net to be installed on any machine which will initiate a database connection (specifically, any machine that will run a DBSession object—for example, a DBSession service object).

Database on same node as partition with DBSession object — The first model installs the Forte server partition on the same node as the Oracle RDBMS; in this scenario, SQL*Net is not required on either node. This model provides efficient and fast communication between the application and the Oracle database.

Database on any node in network using SQL*Net — The second model allows the Oracle RDBMS to reside on any machine in the network, provided that SQL*Net is running on that machine. This model provides more flexibility.

The node on which the database is installed can even be a different operating system and hardware platform than the node running the Forte application, as long as both the Forte server and the Oracle server support the same network transport protocol (TCP/IP, for example) and SQL*Net is installed on the Forte server machine.

Rdb Notes

Forte can access both local and remote Rdb databases from OpenVMS systems. Forte can also access non-Rdb databases through Oracle's Database Integrator (DBI) family of products. Note that Forte requires only the Rdb Run-Time license to access Rdb databases.

Because of the design of the Forte architecture, the connectivity model departs from conventional Rdb connections. In the conventional configuration of a local, native Rdb application, the Rdb runtime system runs within the server partition and accesses Rdb data on the server node.

In a typical Rdb client-server configuration, SQL/Services enables the client application to access Rdb data on the server node; SQL/Services software is required on both sides of the connection. DECnet, TCP/IP, and AppleTalk communication protocols are supported between SQL/Services clients and servers.

Communication handled by Forte

In a Forte client-server application accessing local Rdb data, communication between the client and server is handled by Forte rather than SQL/Services. If the Forte server partition is installed on the same node as the Rdb database, Forte will make an efficient local database connection.

The Rdb database can be installed on an OpenVMS machine that does not have Forte software installed. The Forte database service object accesses the remote Rdb database using DECnet and an RDB$REMOTE account. While less efficient than the configuration described above, this illustrates the flexibility available with the Forte/Rdb interface.

Unsupported Database Features

Most database management systems extend the ANSI standard SQL language. Whenever possible, Forte supports vendor-specific SQL extensions; however some extensions are inconsistent with the implementation of TOOL SQL and cannot be supported. These are indicated below.

- scroll cursors (cursors that can be fetched in a non-next fashion)

  For Informix only, scroll cursors are supported through the DBSession class interface; scroll cursors are not supported for other databases.

- Informix *hold cursors* are not supported (cursors that remain open across transactions).

Using ODBC

Microsoft's Open Database Connectivity (ODBC) is a standard application programming interface (API) for accessing data stored in database management systems. Using ODBC, you can develop, compile, and ship an application that is database independent. Using standard SQL to access data, an ODBC application can access various databases without modification or recompilation. At runtime, dynamic-link libraries, called *database drivers,* are linked on demand to access a specific *data source* through a specific communication method.

The ODBC architecture has four main components, shown in Figure 3 and described below.

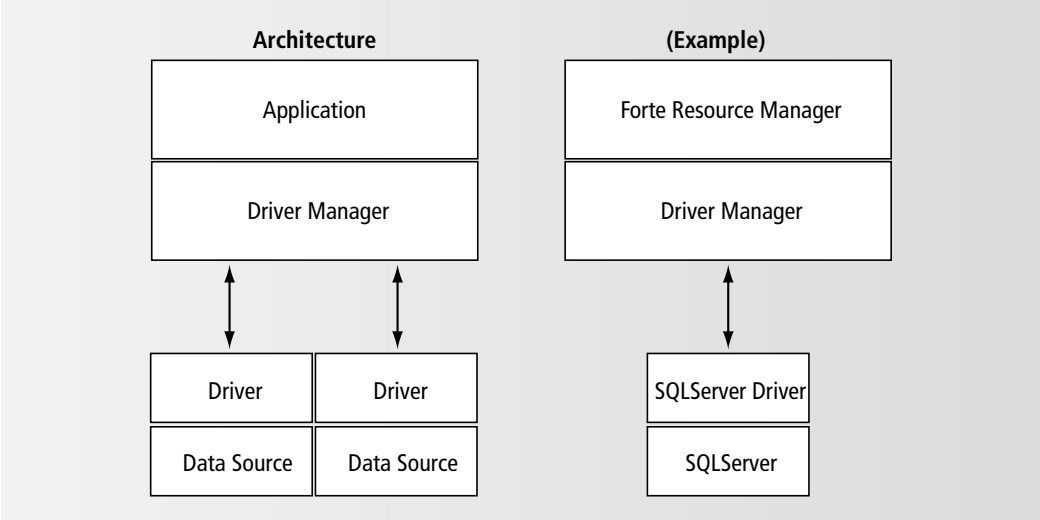


Figure 3 *ODBC Architecture*

application Performs processing and calls ODBC functions to submit SQL statements that manipulate and/or retrieve data in the target database. When using Forte with ODBC, the Forte ODBC resource manager is the ODBC application. This component is the only one provided by Forte; all other components must be in place for a Forte application to use ODBC.

driver manager Loads database drivers on behalf of an application. This component allows ODBC applications to access different databases without recompilation. The driver manager is provided by Microsoft on Windows platforms.

database driver Processes ODBC function calls, submits SQL requests to a specific data source, and returns results to an application. Database drivers are provided by database vendors or other third parties.

data source The user's data, the associated operating system, database, and network platform (if any) used to access the DBMS.

Supported Data Sources

To see the official list of which data sources (and versions) Forte certifies against which platforms, refer to the platform support matrix, which is available at http://www.forte.com/support/platforms.html.

Supported Database Drivers

In addition to a data source, you must have a supported database driver for that data source. The platform support matrix, at http://www.forte.com/support/platforms.html, lists certified database drivers for each platform.

Because various databases and database drivers offer a range of functionality, the ODBC interface defines conformance levels to categorize the level of support each driver and database provides. Forte applications require drivers to support:

- Level 1 API and Core SQL grammar, as defined in the ODBC 2.1 SDK.

Also, Forte can use ODBC array interface support if the ODBC driver supports:

- SQL Extended Fetch, so that Forte can perform an array fetch
- SQL Param Options, so that Forte can perform array inserts

If the driver does not support these features, Forte simply does not use the array fetch and insert features.

Using ODBC When Running Forte in Standalone Model

The primary benefit of ODBC is that it allows you to access a database system when running Forte standalone, rather than distributed. This can be useful in a number of situations, for both application developers and end users.

For example, an application developer can prototype an entire distributed Forte application running standalone on Windows running against a SQL Server data source. Later, the developer could deploy the application in a different environment against other, possibly multiple, databases.

Another example is an end user who daily accesses a local database while running standalone, but periodically connects to a distributed environment to access a different database. Such a user could use ODBC when running standalone, regardless of the type of database he accesses when running distributed.

When the Forte development/runtime system is run standalone, Forte always assumes a Database Resource manager type of ODBC. For more information on using ODBC in standalone mode, refer to "Using a Local Database when Running Forte Standalone" on page 35.

Cautions on Using ODBC with Forte

When you use ODBC with Forte you should be aware that not all data sources, nor all drivers, provide the same functionality. Depending upon which data source and driver you use, some database features may not be available.

ODBC drivers must be thread safe

On NT and Windows 95, your ODBC driver must be thread safe. Forte exploits the native threading capability on these platforms and requires the ODBC driver to operate in a multithreaded environment.

Data source limitations

If a Forte application attempts to use a database feature that is not supported by a particular data source, Forte generates an exception (a DBUsageException or a DBResourceException, depending upon the circumstances). Examples of features that may generate this type of exception include outer joins and database procedures.

Chapter 2

Defining a Resource Manager

Before a Forte application can access data in a relational database, you must create a *resource name* for the database. This chapter contains the following topics:

- the purpose of the database resource manager
- choosing a node for the resource manager
- defining a database resource name
- running the DynamicSQL example to test a resource name
- accessing a local database when running Forte standalone

If you are a Forte system manager, follow the instructions in this chapter to set up and test a resource name.

If you are a Forte application developer, ask your system manager for valid resource names. Then, to access a database from a Forte application, you must establish database connections as described in Chapter 3, "Making a Database Connection."

About Resource Managers

For a given database to be accessible by a Forte application, a *resource manager* must be associated with the database. A resource manager simply indicates a *type* of database (as specified by the name of the database vendor, which indicates the database's interface) that can be accessed from a particular *node*. Note that a resource manager does *not* uniquely identify a particular database instance, but only a database type, such as Oracle, Sybase, or ODBC.

Resource name points to resource manager

The Forte system manager creates a database *resource name* using the Environment Console. Each resource name is associated with a resource manager. See "Defining a Resource Name" on page 28 to see an example of the dialog window. Forte application developers use the resource name in applications that need to access a database of that type (vendor). During partitioning, Forte places the database service objects on the same node as the resource managers they use. Subsequently, all database connections to a given database are established through the resource name for that database.

A resource name contains no connection information (specifically, unique database name, username, and password). All connection information is provided either through a DBSession service object, or when connecting to a database using the ConnectDB method on the DBResourceMgr class. See Chapter 3, "Making a Database Connection" for a discussion of database connection options.

One resource name can be used by multiple database service objects. Each service object would reference the same resource name, but could use different database names when providing the connection string to initiate a database session. Figure 4 shows an example of three different applications using the same database resource name but connecting to different databases.

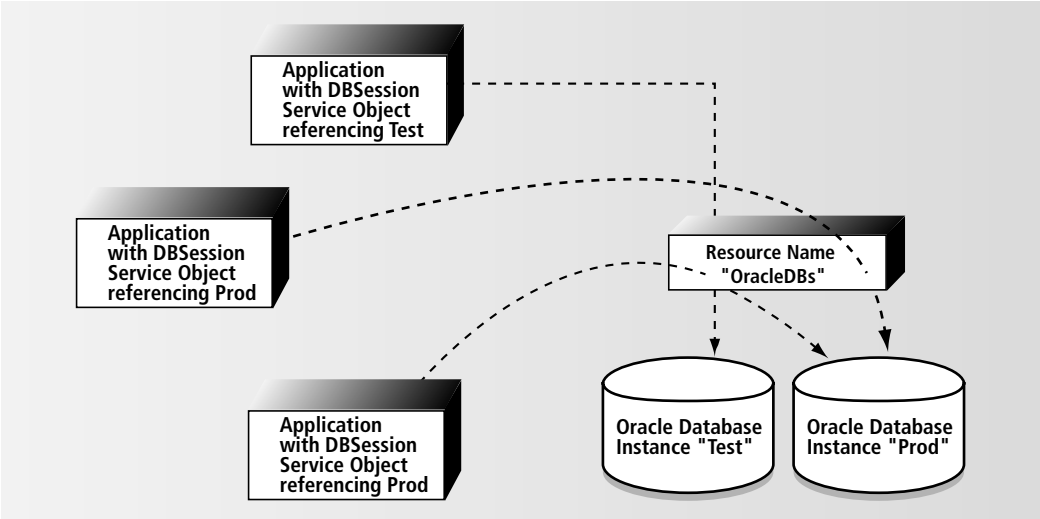


Figure 4 *Resource Name used by Two Databases*

Distributed development and access

The Forte system manager defines a resource name for access by multiple application developers (and end users) running Forte in distributed mode. To do so, the manager uses the Node Properties dialog of the Environment Console, and defines the resource name on a node on which the DBMS client interface resides. The node need not have the database itself installed, as long as the node can access the database (for example, any database-specific environment variables are set). "Defining a Resource Name" on page 28 describes this procedure.

Stand-alone development and access

Any Forte application developer or end user running Forte in standalone mode uses an ODBC resource name by default; all database connections are assumed to use ODBC. The name of the data source must be specified, but the resource name and resource manager are assumed. See "Using a Local Database when Running Forte Standalone" on page 35 for more information.

Choosing a Node for the Resource Name

You should define a resource name on a node that has guaranteed and efficient access to the target database system or ODBC data source. When choosing the node, keep the following in mind:

- The Forte database interface can be considered the client side of a specific database vendor's client-server support. This is the case even though Forte generates server partitions for the DBSession or DBResourceMgr service objects.
- By default, Forte partitioning locates service objects on the same node as the resource manager they use.
- Some database vendors provide client-server connectivity using standard networking protocols or fast transfer using shared memory when the client is on the same node as the DBMS server. It may be advantageous, therefore, to define a resource name on the same node that runs the DBMS server process(es).
- For databases accessed through the ODBC interface, the resource name must be defined on a Windows node that contains the required ODBC driver manager and drivers.
- It can sometimes be advantageous to load balance client-server activity by having clients run on a node (or nodes) other than that which runs the server process(es).

Defining a Resource Name

Skip this step if using ODBC in stand-alone mode

If you are running Forte standalone using an ODBC data source, this section is irrelevant because you do not use an Environment Console in standalone mode). However, if you are a developer who runs both stand-alone and connected (accessing both local and central databases), then your Forte system manager would complete these steps to define a resource name for a server-resident database.

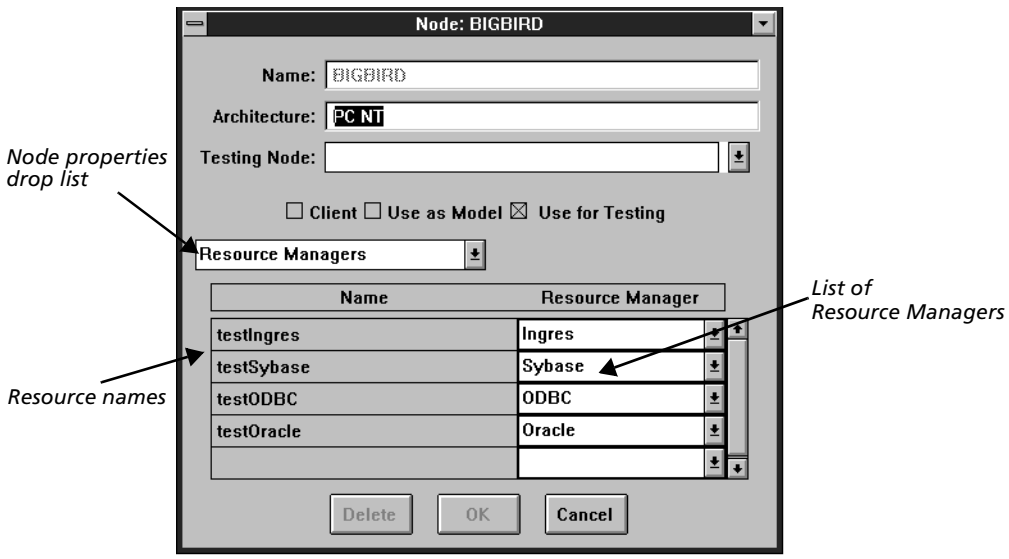
After you choose a node on which to define the resource name, you can use either the Environment Console or the Escript utility to define a resource name.

To define a resource name for a database using Environment Console:

1. Open the Environment Console.
2. Choose either the **View** > **Node Topology** or **View** > **Node Outline** command.
3. Lock the environment by clicking the Locked toggle.
4. Select the node on which you will define the resource name.

   It is from this node that connections will be made to the database or ODBC data source. Therefore, this node must be running either the database system itself or the database client software.
5. For this node, choose the **Component** > **Properties...** command.

   The Node Properties dialog appears, as shown below. Make sure that Resource Managers, the default choice, appears in the node properties drop list.



6. Enter any unique name to be the resource name for the database.

   All resource names defined for a given environment must be unique within that environment (the name space is a Forte environment).

   You will use this name later, in the “Database Manager” field of the Database tab page of the Service Object Properties dialog (see “Entering Database Information” on page 40). Because the resource name is not physically mapped to any specific database, you can easily move the resource name to a different node, if desirable, without changing any Forte code.

7 Choose the appropriate database type (vendor) from the Resource Manager drop list.

   The drop list varies by platform; it shows only database interfaces that Forte supports for that node's architecture. It does not reflect databases that are actually installed on the node.

8 Save the changes to the environment using the **File > Save Environment** command.

9 Unlock the environment by clicking the Locked toggle. Exit the Environment Console.

10 If necessary, set the appropriate environment variables for the database, as described later in this chapter. On some platforms, notably VMS, other environmental considerations such as process privileges and quotas may apply.

Using Escript to Define a Resource Name

Rather than using the Environment Console, you can use the Escript command **AddExternalRM** to define a resource name. For more information on the Node Properties dialog, or the AddExternalRM command, refer to the *Forte 4GL System Management Guide*.

Environment Variables

Installation sets all required environment variables

When you install or reinstall Forte on a server node, you provide database-specific information that Forte uses to automatically update all required environment variables for each database. So, you may not need to manually set any environment variables, because the installation does so for you. However, if you want to add a new database (that is, a new database instance from the same vendor, or a database from a different vendor) then you must update the environment variables so that Forte can interact with the new database. Refer to the appropriate sections below to set any environment variables that apply to the database you are adding.

Environment variables at three different levels may affect your Forte database application. You can set environment variables (or logical names) required or used by (1) the operating system, (2) by Forte, or (3) by the database.

O/S environment variables

You should set the UNIX LIBPATH environment variable to include the library directory for each database you will connect to on a UNIX node. Check either of the files fortedef.sh (for K or Bourne shell) or fortedef.csh (for C shell) and add the value for LIBPATH if necessary. The LIBPATH variable must be set before you start the Forte Node Manager process.

The actual name for LIBPATH varies slightly by platform:

Platform	Name of LIBPATH to use
Compaq Tru 64 Unix, Solaris, DG, PTX	LD_LIBRARY_PATH
RS6000	LIBPATH
HP	SHLIB_PATH

Forte environment variables

Forte requires no additional Forte-specific environment variables to access databases. However, there are a few optional environment variables you can use to tune application performance (FORTE_DB_MAX_STATEMENTS and FORTE_DB_VENDORFLG are two that relate to databases; for more information on these variables see the Forte online Help).

Vendor-specific environment variables

All database vendor-specific environment variables must be set in the Forte server process containing the database resource manager or session. Since Forte server partitions inherit environment variables from their Forte Node Manager, it is sufficient to set the database environment variables in the Forte Node Manager process before any Forte process is started.

The following table lists the database vendor-specific environment variables that must be set for a given vendor-type of database. These variables are described in following sections.

Database	Environment Variables
DB2	DB2INSTANCE and other variables specified in the files db2profile and db2cschrc
Informix	INFORMIXDIR, INFORMIXSERVER
Oracle	ORACLE_HOME, ORACLE_SID
Rdb	RDMS$RUJ, SORTWORK, and RDMS$DEBUG_FLAGS
Sybase	SYBASE, if root directory (interfaces file) is in non-standard location.

DB2 Variables

A Forte Node Manager may access only one DB2/6000 instance. Each DB2/6000 instance has one of two example files, db2profile (for K shell) and db2cschrc (for C shell), which defines the environment variables necessary to access the instance (such as DB2INSTANCE). You should incorporate (add) the appropriate file into either your .profile file (for K shell) or .cshrc file (for C shell). See IBM's *DATABASE 2 AIX/6000 Installation Guide* for a description of these environment variables.

Informix Variables

Informix uses a number of environment variables to modify a given installation. Two variables are particularly important for Forte applications:

INFORMIXDIR A required variable that specifies the top directory of the node's Informix product installation.

INFORMIXSERVER A required variable that can be used to specify a default Informix database server for database connections. When a connection resourceName parameter does not contain a server name component, the value set for INFORMIXSERVER is used.

Environment Variable	Platforms	Setting
INFORMIXSERVER and INFORMIXDIR	Solaris, Alpha OSF, DG Intel, PTX Intel	LD_LIBRARY_PATH should include $INFORMIXDIR/lib:$INFORMIXDIR/lib/esql
	RS6000	LIBPATH should include $INFORMIXDIR/lib:$INFORMIXDIR/lib/esql
	HP9000	SHLIB_PATH should include $INFORMIXDIR/lib:$INFORMIXDIR/lib/esql

Oracle Variables

ORACLE_HOME During the Oracle installation, you set the environment variable ORACLE_HOME to indicate the home directory for the Oracle installation. SQL*Net V2 uses this environment variable to locate the file tnsnames.ora.

ORACLE_SID You should also set the environment variable ORACLE_SID to the instance of Oracle to which you are going to connect. Local connections to an Oracle database require ORACLE_SID to be set.

Rdb Variables

While there are no VMS logical names required to access Rdb databases from Forte, some Rdb-specific logical names may be useful. Logical names can be used for two distinct purposes:

- Rdb configuration and load balancing, using the RDMS$RUJ, SORTWORK, and RDMS$DEBUG_FLAGS logical names.
- Naming databases, by defining a logical name that translates to an Rdb database specification. This logical name can then be specified in the ConnectDB method of the DBResourceMgr class (see "Connecting to a Database" on page 38). Using a logical name for a database makes your Forte code and service object definitions more portable.

For a Forte/Rdb session to process a logical name, the name must be "visible" to the Rdb server process created by the Forte Node Manager. The recommended procedure is to define the name in the Forte "global logical name table" by adding the definition of the logical name to the FORTE_STARTUP procedure. See the section on installing on Alpha OpenVMS in the *Forte 4GL System Installation Guide* for more details.

Sybase Variables

During the Sybase installation, a file named "interfaces" is created in the root directory for Sybase. By default, this directory is the login directory for the Sybase account. Sybase CB-Library must be able to find the interfaces file to translate the server name to an appropriate network address. If the Sybase root directory is not in the login directory for the Sybase account, then you must use the environment variable **SYBASE** (a logical name on VMS) to tell Sybase where to find the interfaces file.

Removing a Resource Name

You might wish to remove a resource name from one node in an environment so that you can redefine it on another node. You might do this if you moved your database to a more powerful machine. You could redefine the resource name to use a resource manager on the newer machine; this would cause Forte to place the servers that use that resource manager on the newer machine during partitioning.

To remove a resource name, complete steps 1 through 4 as described in "Defining a Resource Name" on page 28. Then, highlight the resource name you wish to remove and click the Delete button.

Testing a Resource Name with the DynamicSQL Example

You can use the Forte sample application called DynamicSQL to test a new or existing resource name. This section contains instructions for running DynamicSQL. When the instructions differ by database vendor, follow the instructions that pertain to your particular vendor.

The DynamicSQL example does not create any new tables or add any data to your database. It simply opens a Forte window through which you can enter any SQL statement using tables and data that already exist in the database.

To test a resource name:

1 Using the vendor's utilities, verify that the database (or ODBC data source) is accessible. After you complete this step, your database should also be accessible from Forte.

Database Vendor	To verify that the database is accessible:
DB2	Login to the node. Define the environment variables appropriate for accessing the local DB2/6000 database "dbname." Run the DB2 command line processor ("db2") and use the following command: **connect to** *dbname* **user** *uname* **using** *upass*
Informix	Login to the node using the uname account. Define the INFORMIXDIR environment variable, and if appropriate, INFORMIXSERVER. Use the **dbaccess** command to connect to the Informix database.
ODBC	Login to the node and use the database-specific interface to test database access. For example, if you are using a Microsoft SQL Server data source, you would run the isql command line processor and connect to the dbname database, using *uname* and *upass*.
Oracle	Use the following command: **sqlplus** *scott/tiger@servicename* When you should see a SQL> prompt, type **Quit** to exit the program.
RDB	Use the following command: $ RMU /SHOW VERSION You should see the message "Executing RMU for DEC Rdb Vx.y." If your database node has an Rdb Interactive or Development license, then you can verify that the Rdb database is accessible using the SQL$ Interactive SQL program. For example, if the database filename associated with MYDBNAME is DKAO:[DATABASES]FINCHES, then enter: $ RUN SYS$SYSTEM:SQL$ SQL> ATTACH 'FILENAME DKAO:[DATABASES]FINCHES': SQL> SHOW TABLES SQL> EXIT
Sybase	Use the following command: **isql** -S *DBname* **-U** *uname* **-P** *upass*

2 Create a resource name for the database (vendor) of your choice.

If you already have an existing resource name, go to Step 3.

If you do not have an existing resource name, follow the instructions in the previous section. The DynamicSQL example assumes that the resource name is "AnyDBMgr." You can either use AnyDBMgr, or modify the reference to the resource name to match your name (described in the next step).

3 Create the DynamicSQL project. Open the Repository Workshop and import the two files:

$FORTE_ROOT/install/examples/frame/utility.pex

$FORTE_ROOT/install/examples/database/dynsql. pex

If you have an existing resource name, you must edit the service object for this example to use your resource name. In the Project Workshop for the DynamicSQL project, double-click on the AnyDBMgr service object. In the Service Object Properties dialog (General tab page), change the name in the Database Manager field from "AnyDBMgr" to match your resource name.

4 Run the DynamicSQL application.

The application input and output appear in the console/trace window, so you should open the window if it is iconized.

5 You are prompted for a database name, a username and a password. Respond to the prompts using the format that corresponds to your database vendor, as described in the following table.

Database Vendor	Database	Username	Password
DB2	*dbname* The name of the local DB2 database, as specified on a DB2/6000 SQL CONNECT statement. This database must be accessible from the local instance defined by the DB2INSTANCE environment variable. Example: TestDB2DB	*uname* An authorization name associated with a DB2/6000 SQL CONNECT statement. *uname* must be an authorized login user name on the AIX system.	*upass* The valid AIX password for *uname*.
Informix	*dbname@dbserver* The name of the Informix server and database. or *dbname* The name of the Informix database. The translation of the INFORMIXSERVER environment variable is used for the Informix server. Example: testinfdb or testinfdb@HQ3	*uname* A valid UNIX account on the Forte server node.	*upass* The password associated with the *uname* account.
ODBC	*dbname* The name of a valid ODBC data source. Example: MyLocalDB	*uname* *uname* must follow the conventions related to the database to which you are connecting.	*upass* *upass* must follow the conventions related to the database to which you are connecting.
Oracle	@[*TNSResourceName*] A database name, as specified in the SQL*Net V2 file ORACLE_HOME/network/admin/tnsnames.ora If *TNSResourceName* is omitted, then the connection is to a local database and the environment variable ORACLE_SID must be set. *TNSResourceName* is case-sensitive. Example: @testora or @	*uname* A valid username for the current Oracle database instance.	*upass* The password for the Oracle username.
Rdb	RDB.*dbname* *dbname* is the Rdb database filename specification used to attach to the database from the current node. A logical name can be used only if it is "visible" to the Forte process associated with the RDBMGR service. The Rdb connection established by Forte runs under the persona of the specified VMS user. This persona includes the username, UIC, authorized privileges and (new to Forte Release 3), any general identifiers associated with the user. The Rdb database should have appropriate access granted to the user's UIC and/or identifiers. Example: RDB.MYDBNAME	*uname* A valid VMS username.	*upass* The password associated with the *uname* account.
Sybase	resource.*dbname* resource is the name of the resource in the Sybase interfaces file. dbname is the name of the database for that server. Example: SYBASE.master	*uname* A valid Sybase username.	*upass* The password associated with the *uname* account.

6 To use the application, enter SQL select statements for tables that you know to exist in the database. Type Quit to exit.

(Appendix A, "Database Example Applications" also contains instructions for installing and running the DynamicSQL example.)

Using a Local Database when Running Forte Standalone

You can run Forte in standalone mode on Windows platformsForte (as opposed to distributed mode) and connect to a local database using the ODBC database interface. Using this configuration you can run standalone Forte database applications, or develop portable database applications that can later be deployed in distributed systems running on multiple platforms and accessing multiple databases from different vendors.

When you run Forte standalone, an ODBC resource manager is always assumed. Forte automatically assumes a resource manager type of ODBC for any DBSessions or DBResourceMgr service objects; you simply need to enter valid, local data source name when you define a database service object. For additional information about running standalone refer to "Entering Database Information" on page 40.

Setting up a Forte ODBC Data Source on Windows

This section contains instructions for making an ODBC data source available to a Forte system running standalone on any Windows platform.

Local or remote data source

These instructions assume that you have already installed the underlying data source (database). They allow you to define a data source that is running either locally on the same machine or remotely; in either case Forte standalone can use it as a data source.

To define a new ODBC data source:

1. Install the ODBC Driver Manager (this is a component of ODBC, not the driver itself).
2. Install the appropriate database driver for each target database (each database that you wish ODBC to access for your Forte application).

   For a review of the ODBC architecture, showing the Driver Manager and database drivers, see Figure 3 on page 23. Also, refer to the *ODBC 2.0 Programmer's Reference and SDK Guide* for additional information about installing these components.
3. Open the ODBC Administrator from the ODBC Control Panel. Click on the Add button. You will see a list of installed drivers, from which you select one for the new data source.
4. Choose the appropriate driver for your target data source and click OK.

   To see a list of installed drivers, you simply click on the Drivers button on the ODBC Administrator dialog.

   For example, to create a data source for a local SQL Server database, choose SQL Server as the driver type.
5. In the next dialog that appears, the ODBC Data Source Setup, enter information about the Data Source.

   You can specify any database accessible to this node, whether local or remote. You can even specify a flat file as a data source using ODBC.

   Enter any arbitrary name; you will use this name later as the Database Name in the Service Object Properties sheets, when specifying connection information.

   Enter appropriate information for the Server, the Network Address and Network Library. At this point you can specify either a local or remote data source.

   When you click OK, you have added a new data source.

After an ODBC data source has been defined to ODBC, there are a few additional steps required to make the data source available to Forte users and developers.

To make the data source available to Forte users:

1. The data source (for example, a database) must be running. From the program group for the database, start the database service by double clicking on the database icon and selecting Start.
2. (If you want Forte users who are running distributed to be able to access the ODBC data source) In the Environment Console, add an ODBC resource manager for the node on which you defined the data source.

   (For detailed instructions see "Defining a Resource Name" on page 28.)

Using an Existing Data Source

To see the names of existing ODBC data sources, open the ODBC Administrator. You will see a display containing the names of valid data sources with the drivers they use.

To see the available drivers, press the Drivers button in the ODBC Administrator.

Chapter 3

Making a Database Connection

This chapter describes options you have for connecting to a database. Many applications will use a service object to initiate a database connection. Both types of database service objects, DBSession and DBResourceMgr, allow you to connect to a database, establish a database session, and query and update database data.

You can also connect to a database without using a database service object. This process is also described in this chapter.

This chapter contains the following topics:

- choosing between the two types of database service objects
- creating a service object and connecting to a database
- connecting to a database without a service object
- using connection options, such as creating a default DBSession service object (a default user name and password) and using variable user names and passwords
- resuming dropped database connections

Connecting to a Database

You connect to the database by establishing a database session. A database session is a connection to a particular database (using a Forte database resource name) through which you execute an isolated set of statements.

You can open and use any number of sessions to a single database, subject to operating system and database limitations. For example, to avoid creating a long-term transaction when an end user is browsing through the database and making changes, you can use two different database sessions. The first session can open a cursor for read only, and the second session can use short transactions to make any database updates that the end user requests. You can also use sessions to access more than one database or type of database in the same Forte application.

To initiate a database session you specify connection information (a database and a valid username and password). You can start a database session in three ways:

- using a DBSession service object (based upon the DBSession class)
- using a DBResourceMgr service object (based upon the DBResourceMgr class)
- using no service object

These three ways are described in the following sections.

Each connection has a DBSession object

Every database connection (session) is represented by an object of class DBSession. When you invoke the ConnectDB method on a DBResourceMgr service object, the ConnectDB method returns a DBSession object. In fact, Forte uses the same ConnectDB method for every database connection, but the method is transparent when used with a DBSession service object or without any service object at all. Subsequently, your application can execute TOOL SQL statements or invoke methods from the GenericDBMS library during a database session; both cases cause Forte to invoke methods on the DBSession class on the current DBSession object.

Every SQL operation is associated with a session

Every SQL statement issued against a database is associated with an **on session** clause that identifies an active database session. You can use a service object name or reference to an object of the DBSession class. You can also omit the **on session** clause to specify the default session called DefaultDBSession (see “Creating a DefaultDBSession Service Object” on page 50).

Using Database Service Objects

Typically you will use either a DBSession service object or a DBResourceMgr service object to initiate connections to a database, although you can make connections without a service object (see “When You Should Not Use a Service Object” on page 39). The two service objects are used in different circumstances, as described below.

(See “Creating a Database Service Object” on page 40 for instructions for defining either type of service object.)

When to Use a DBSession Service Object

DBSession service objects are perhaps the most convenient way to access a database, but offer somewhat less flexibility than the DBResourceMgr service. You can use a DBSession service object if the following are true:

- You can pre-specify the connection information (resource name, database name, user name and password) when you define the service object.

  (You can override user information using environment variables; see “Using Variable User Names and Passwords” on page 49.)

- Your application does not require a dynamic number of database sessions. While there can be multiple sessions, the number of sessions cannot change at runtime.

Advantages

The DBSession service object has the following advantages:

- It automatically connects to and disconnects from the specified database, using the pre-specified connection information, at application start-up and shutdown.

Disadvantages

The DBSession service object has the following disadvantages:

- You must provide the user name/password and server/database name when you partition the application that contains the service object. (You may not specify these items after partitioning or when the application is running.)
- If the service object's visibility is set to environment, then many clients share the same DBSession service object; thus, you cannot use individual users' accounts for the sessions. Instead, you must use a generic account that stays the same across transaction boundaries.

When to Use a DBResourceMgr Service Object

You must use a DBResourceMgr service object if you want your application to obtain connection information at runtime.

Advantages

The DBResourceMgr service object has the following advantages:

- You can invoke the ConnectDB method multiple times to start multiple database sessions.
- You can use different connection information for each database session.

Disadvantages

The primary disadvantage to using a DBResourceMgr service object is that you must explicitly manage the connects and disconnects in your program. Some vendor-specific limitations do exist for DBResourceMgr service objects:

Rdb All concurrent sessions within a DBResourceMgr must use the same user name.

Oracle on VMS Only one session can be directly connected to the Oracle RDBMS; other concurrent sessions must be connected via the SQL*Net mailbox driver.

Informix Only a single shared memory connection may be active in a Forte partition. Multiple network connections to Informix databases are fully supported.

When You Should Not Use a Service Object

In many applications, all database connections are made through a database service object. However, a characteristic of any database service object is that it is associated with one database when it is defined, and can only be used to connect to that particular database. This characteristic of service objects means that any application that must be able to connect to multiple databases cannot use either type of service object. For example, many applications might reasonably prompt users to specify which database they wish to connect to when they start the application.

You can write an application that is not bound to one database. Rather than using a service object to initiate a database session, you invoke the Connect method on the DBSession class, after you have determined which vendor-type of database the user will connect to. The vendor type of library must be determined first, because all objects and methods that Forte uses during a given database session are actually vendor-specific.

See "Connecting to a Database without a Service Object" on page 47 for detailed instructions on initiating database sessions without service objects.

Creating a Database Service Object

A project can contain any number of DBResourceMgr or DBSession service objects. To create a new service object, use the Project Workshop (see *A Guide to the Forte 4GL Workshops* for details). Then use the **New > Service Object** command. The Service Object dialog appears, as shown in Figure 5.

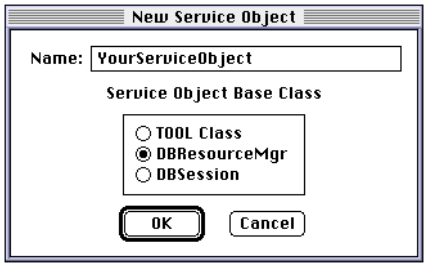


Figure 5 *New Service Object Dialog*

Specify a name for the service object and indicate whether it is a DBSession or DBResourceMgr service object. The Service Object Properties dialog appears next.

The Service Object Properties dialog uses tab pages to define the properties of one service object. A DBResourceManager uses three tab pages while a DBSession service object uses four pages.

To see more information on using the Forte workshops, or the procedures and dialogs you use to create service objects, refer to *A Guide to the Forte 4GL Workshops*.

Entering General Properties

Both types of service object use the General Properties tab page, shown in Figure 6.

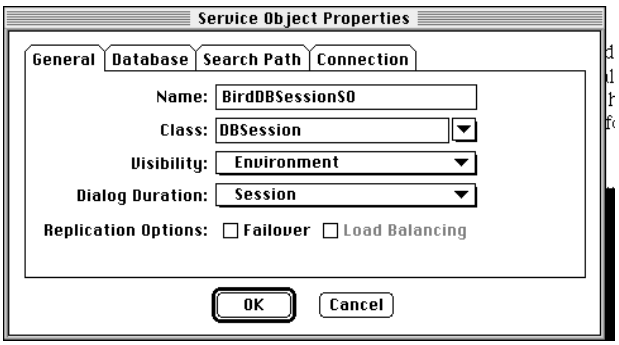


Figure 6 *General Properties Tab Page*

Visibility

The default visibility of database service objects is environment-visible.

Dialog Duration

If you want the service object to be replicated for failover, load balancing, or both, then you must use either Message or Transaction dialog duration; you cannot use Session duration with replication.

Replication Options

Check one or both of the fields “Load Balancing” or “Failover” if you wish the service object to have these characteristics. For more information see the *Forte 4GL Programming Guide*.

Entering Database Information

Both types of service object use the Database tab page, although the fields differ for the two types of objects.

For both types of service objects you specify the resource name to be used for database connections initiated by this service object. Use the drop list to enter this information in the “Database Manager” field.

For a DBResourceMgr service object, this tab page shows only the “Database Manager” field. Only the database manager information is required, because this type of service object allows different user information to be specified whenever a database connection is initiated.

For a DBSession service object, three additional fields appear as shown in Figure 7. You must enter user information for a DBSession service object, as it uses the same user information whenever it requests a database connection.

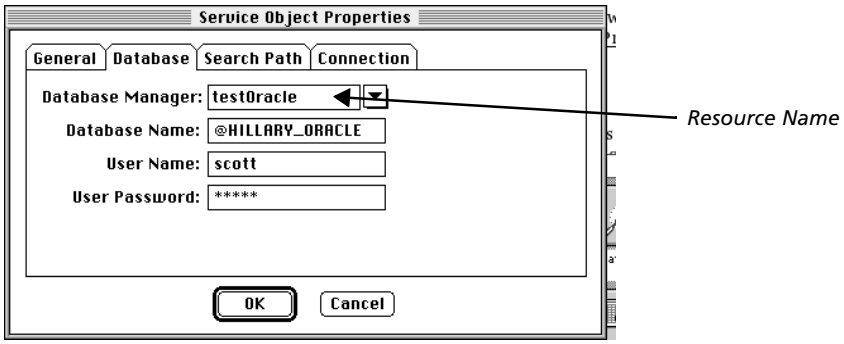


Figure 7 *Database Tab Page*

DatabaseManager

The drop list for this field shows the resource names that have been defined by the Forte system manager (as described in “Defining a Resource Name” on page 28.) Select the resource name for the type (vendor) of database with which this service object will interact.

If you are running Forte stand-alone, the drop list is empty. It does not matter what you enter, because Forte assumes an ODBC resource manager; however, you must enter something. This name is not case sensitive; in fact, you do not use it anywhere else.

Database Name

Enter the name of the actual database with which the service object will interact.

Vendor	Format	Description
DB2	dbname	*dbname* is the name of the DB2 database as known to the DB2 instance (defined by the DB2INSTANCE environment variable) running on AIX.
Informix	*dbname*[@*dbservername*]	If *dbservername* is not specified, then the Informix environment variable INFORMIXSERVER must be set to identify the default server.
Oracle	@[TNSResourceName]	The *TNSResourceName*, including the @ symbol, is passed, unedited, to the Oracle connection routine. If only the "@" symbol is specified, then a local connection is made to the instance associated with the Environment Manager's ORACLE_SID value.
ODBC	data source name	Specifies the data the application will access, and its associated operating system, DBMS, and network platform (if any). Data source names are case sensitive.
Rdb	*DBtype.databasename*	*DBtype* is the type of database to which you are connecting (see the Rdb table in the Forte online Help) *databasename* is the location of the database.
Sybase	*servername*[.*databasename*]	*servername* is the SYBASE domain server name. If you do not specify the database name, Forte uses the Sybase default database for the user.

Username and Password

Enter a valid username and password for the target database. Case sensitivity of these two fields depends upon the target database.

Entering Search Path Information

Both types of service object use the Search Path tab page, as shown in Figure 8. Refer to *A Guide to the Forte 4GL Workshops* for a discussion of how the search path is specified and used.

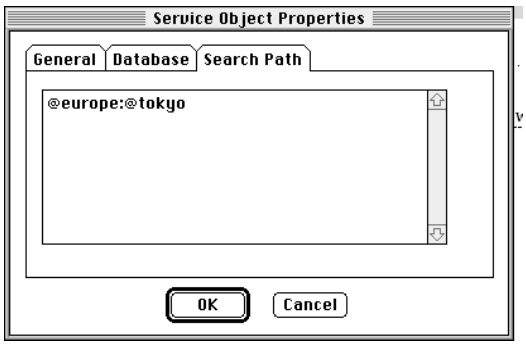


Figure 8 *Search Path Tab Page*

Entering Connection Information

Specifying connection options

Only a DBSession service object uses the Connection tab page, shown in Figure 9. This page is entirely optional. Use this page if you want to specify connection options for any database connection to be made by the DBSession service object.

To specify each connection option you use a name-value pair. To see a list of valid options, refer to the description of the optionList parameter for the Connect method (see the Forte online Help).

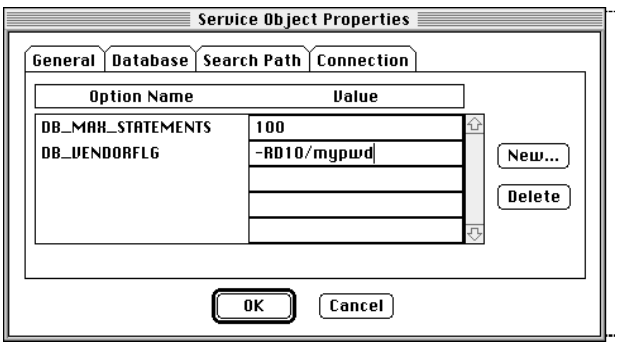


Figure 9 *Connection Tab Page*

You can specify Ingres connection flags by using the DB_VENDORFLG option. For example, to specify a role to use for an Ingres connection, you would enter DB_VENDORFLG under OptionName, and **-R***roleID/password* under Value.

Specifying connection options in a method

While only the DBSession service object allows you to enter connection options using the Connection tab page, you can enter connection options if you connect using either the ConnectDB method on DBResourceMgr or the Connect method on DBSession. An example follows:

```
options : Array of NamedElement = new;
opt : NamedElement = new (Name= 'DB_VENDORFLG',
   Object=TextData(Value='-RroleID/password'));
options.Append(opt);
myResource.ConnectDB
    (resourceName= 'IngresDB',
    userName='ingres', userPassword="",
    optionList = options);
```

Optimizing Service Object Performance

This section briefly touches upon some considerations that impact performance of your application. It does not discuss a number of designs that can be used to optimize performance of typical types of database applications.

Partitioning and service objects

The location of a DBResourceMgr service object or DBSession service object does impact the performance of your application. By default, Forte automatically places these service objects in a server partition on a node that supports the database vendor being used. However, you can easily change Forte default partitioning.

The most critical factor in determining which partition to use for a service object is "Where is the *user* of the DBSession object with respect to the session?" For best performance the user should be placed in the same partition as the DBSession object.

While the DBSession class is distributed and so can be referenced remotely, doing so will degrade performance. There may be circumstances when an application designer must use a remote DBSession object, but the overall application design must take this into account. Remote access is preferably used primarily during development and testing phases.

This is not to say that data cannot originate from, or be destined for, a remote partition. In fact, this is a typical scenario for a multi-tiered application. The best way to do this is to place a service object responsible for the data manipulation in the same partition as the database and channel access to the database through that service object. Figure 10 illustrates this:

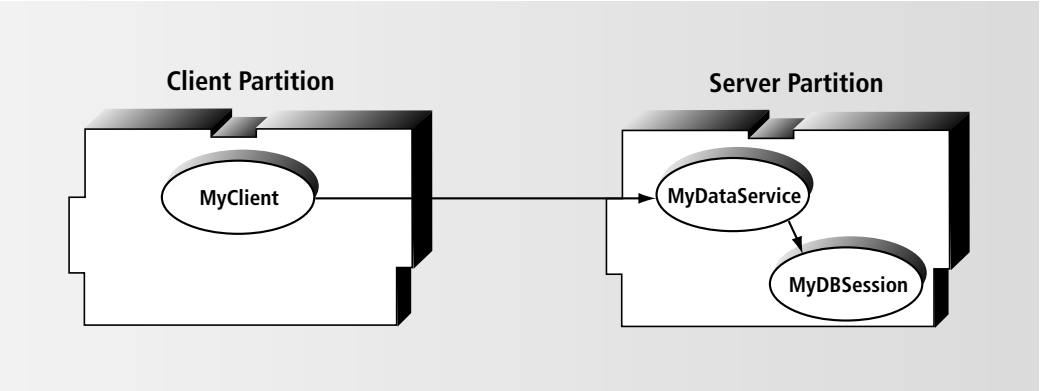


Figure 10 *Partitioning with Intermediate Data Service on Same Partition as Session Service*

Any access to data in the database managed by MyDBSession from MyClient must go through MyDataService. Access to DBSession is from MyDataService which is *local* to MyDBSession. This design optimizes performance, since only the data is sent between the client and server partitions in a single client/server interaction. Whereas, if MyClient accesses MyDBSession directly, the DBSession conversation takes place remotely and requires many more client/server interactions. This design has other benefits:

- MyDataService can do any processing of the data appropriate at this level, potentially resulting in less data to move between the client and server.
- The location of MyDataService can be moved at partitioning time if the database is moved.
- MyDataService can be replicated for fault tolerance or load balancing.

Visibility of the Service Object

If you use an architecture like that shown in Figure 10 you should also set the visibility for the MyDBSession service object to “user” visibility. User visibility means that the service object is only known within its containing partition, not outside. This prevents unintended distributed access; MyClient cannot access or use the MyDBSession service.

On the other hand, the MyDataService object, as the public interface to persistent data, requires “environment” visibility so that it can be seen by MyClient (and any other client). MyDataService would in turn implement this persistence using MyDBSession as the data store.

Making a Database Connection

The following sections describe the standard ways to connect to a database, using either type of database service objects, or connecting to a database specified dynamically at runtime.

Connecting with a DBSession Service Object

A DBSession service object implicitly connects to and disconnects from the database. After you have created a DBSession service object, you simply reference that service object name in a SQL **on session** clause each time you access the database.

To start a database session using a DBSession service object:

1. Define a DBSession service object. (See "Creating a Database Service Object" on page 40.)
2. Use the service object name in the **on session** clause of your SQL statements to specify the database connection the statement should use.

on session clause

In the **on session** clause you can use a service object name or reference to an object of the DBSession class. If you omit the **on session** clause, the default session called DefaultDBSession is used (see "Creating a DefaultDBSession Service Object" on page 50). For example:

Using the **on session** clause

```
sql execute immediate
'create table ArtistTab(Name char(30) not null, Country char(30)
not null)'
on session MySession;
```

The session connects to the database using the user name, password, and database name that were specified when the DBSession service object was created in the Project Workshop. You can override this information at several points, using either the Service Object Properties dialog in the Partition Workshop or the Environment Console.

Connecting with a DBResourceMgr Service Object

One benefit of using a DBResourceMgr service object is that database connection information can be provided at runtime.

To start a database session using a DBResourceMgr service object:

1. Define a DBResourceMgr service object. (See "Creating a Database Service Object" on page 40.)
2. Explicitly invoke the ConnectDB method on the DBResourceMgr service object in your TOOL code.

   Your program may prompt users for database name, user name, and password, as shown in Figure 11. If you do not prompt the user for **userName**, **userPassword**, and **resourceName**, you must enter the actual values to be used for the ConnectDB parameters.

   The ConnectDB method returns a DBSession object, which you will reference when you access the database.

```
MySession : DBSession;
MySession = DBResMgr.ConnectDB(resourceName = DBName,
    userName = UserName,
    userPassword = Password);
```

3. Reference this DBSession object in the **on session** clause of your SQL statements.
4. If desired, invoke the Reconnect method (of the DBSession class) on a DBRemoteAccessException.
5. Explicitly disconnect each DBSession from the database.
6. Set the DBSession object to NIL for optimal memory management.

When the partition that issued the ConnectDB exits, any active DBSessions are automatically disconnected.

Getting connection information

The Forte DynamicDataAccess sample application demonstrates obtaining database and user information before starting a database session. It displays a login screen, shown in Figure 11, in which the user enters a user name, password, and the database to which he or she wants to connect. Once this information is entered, the application invokes the ConnectDB method on the DBResourceMgr service object and connects to the database using the login input as parameters.

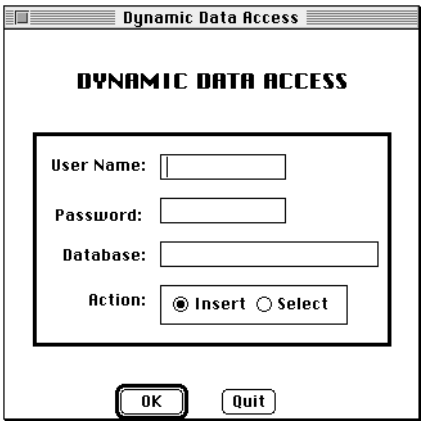


Figure 11 *Login screen for DynamicData Access Example*

For more information about installing and running the Dynamic Data Access example, see "DynamicDataAccess" on page 122.

Connecting to a Database without a Service Object

When you use a DBSession service object or a DBResourceMgr service object, the service object is associated with a specific database resource name during partitioning. For a DBSession service object the user name and password are also pre-specified. So, when you access a database using either type of service object, you must access a predetermined database. If you require more flexibility (specifically, you would like to choose which database to connect to) you do not need to use a service object to initiate a database connection.

The Connect method of the DBSession class allows you to specify, at runtime, connection information (database vendor, database, and user information) to be used to establish the database session. The Connect method also provides some additional security over the use of an environment visible service object.

By using the Connect method and dynamically loading libraries, you can dynamically start a database session without a service object.

Connect requires a DBSession object

The Connect method has the same parameters as, and functions similarly to, the ConnectDB method in the DBResourceMgr class. However, the ConnectDB method creates and returns a DBSession object, whereas the Connect method works on an existing DBSession object.

Connect requires *no* service object

The Connect method does not function on a DBSession service object. The Connect method will only work on a DBSession object created at runtime, either directly as described below or via the ConnectDB method on the DBResourceMgr class.

For reference information on the Connect method, see the Forte online Help.

Dynamically Choosing a Database Vendor

This section describes how to use the Connect method to create and connect a database session. This procedure allows you to indicate the database vendor to be used for the connection (in contrast to when you use a service object).

▶ **To create and connect through a DBSession object:**

1. Load the library that corresponds to the desired database vendor using the FindLibrary method on the Partition object.

   To invoke the FindLibrary method for a database vendor, you must know both the library name and a file key. The vendor's library must be available on the current node. Valid values are shown in the table below.

Database Vendor	LibName	FileKey	ClassName
DB2	DB2	D2	qqdb_DB2Session
Informix	Informix	IX	qqdb_InformixSession
ODBC	ODBC	OD	qqdb_OdbcSession
Oracle	Oracle	OR	qqdb_OracleSession
RDB	Rdb	RD	qqdb_RDBSession
Sybase	Sybase	SY	qqdb_SybaseSession

2. Create a DBSession object of the desired database vendor subclass.

   For this step you must know the name of the DBSession subclass for the desired database vendor. These classes are shown in column 4 of the table above.

3. Connect the DBSession object to the database using the Connect method.

Example

The TOOL code that follows specifies a particular database vendor and creates a vendor-specific DBSession object. In this code you would substitute actual values for the three variables *LibName*, *FileKey*, and *ClassName*; these parameters take string or TextData values.

```
DBVendorLibrary   : Library;
DBVendorClassType : ClassType;
Session           : DBSession;
DBVendorLibrary =
        task.part.FindLibrary(LibName, NIL, 0, FileKey, 4);
DBVendorClassType =
        DBVendorLibrary.FindClass(ClassName);
Session = DBSession(DBVendorClassType.InstanceAlloc());
-- then invoke Session.Connect ...
```

Initially the DBSession object is not connected to the database; the Connect method establishes the actual connection. For example, to establish a connection to an Oracle database you would use the following code:

```
DBVendorLibrary   : Library;
DBVendorClassType : ClassType;
Session           : DBSession;
DBVendorLibrary =
        task.part.FindLibrary('Oracle', NIL, 0, 'OR', 4);
DBVendorClassType =
        DBVendorLibrary.FindClass('qqdb_OracleSession');
Session = DBSession(DBVendorClassType.InstanceAlloc());
Session.Connect(resourceName = '${MY_DATABASE}',
                               userName = 'scott',
                               userPassword = 'tiger');
```

Just as with database service objects, you can use environment variables for the name of the database, as shown in the example (as well as for the user name and password).

Note for DB2

Because the DB2 library is a subclass of ODBC, you must invoke the FindLibrary method twice, first for ODBC and then for DB2. So the example above would require one additional call to use DB2:

```
DBVendorLibrary =
        task.part.FindLibrary('ODBC', NIL, 0, 'OD', 4);
DBVendorLibrary =
        task.part.FindLibrary('DB2', NIL, 0, 'D2', 4);
```

For more information, see the Forte online Help.

Other Connection Information

This section describes connection options, such as specifying a default database session or using default connection information (user name and password). It also describes how to reconnect a dropped session or disconnect a session.

Using Variable User Names and Passwords

Although the DBSession service object requires a user name, password, and database name, you can use environment variables to provide this information at start-up time. You can also reference environment variables at some points in your TOOL code. When Forte encounters a variable, it looks to the environment for the actual value, and replaces the variable accordingly in the TOOL code.

Note

The server portions of the application (which include the DBSession start-up code) are started by the Node Manager process running on the node where the resource name is defined. Therefore, the environment variables need to be defined in the command shell that starts the Node Manager process. Also, depending on the operating system, it may not be possible to change the values being used by the Node Manager once it has started. This is the case on UNIX, for example.

Using environment variables with DBSession objects

You can use environment variables when defining your DBSession service object as a way to dynamically change the database name, user name, and password without altering your application—you simply change the value of the environment variable. Specify the values for these parameters as strings or TextData values containing a dollar sign, followed by an environment variable name surrounded by braces.

For example, you could use the following values for database name, user name, and user password when you define your DBSession service object in the Service Object Properties dialog:

- ${SITE_DATABASE}
- ${SITE_USERNAME}
- ${SITE_PASSWORD}

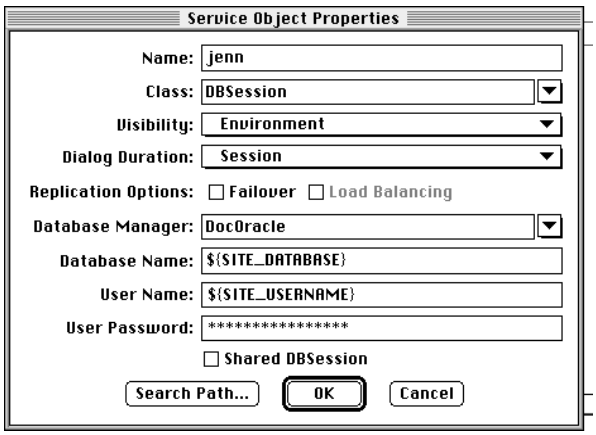


Figure 12 *Using Environment Variables in Service Object Definition*

As Forte processes the values it checks for a dollar sign followed by a name enclosed in braces. If braces are found, Forte checks for an environment variable with the same name. If variables are found, their values are substituted for database name, user name, and user password. For example, your environment could include the following definitions:

- SITE_DATABASE = @Test2Oracle
- SITE_USERNAME = scott
- SITE_PASSWORD = emu

This creates a session for scott/emu on the database Test2Oracle. You can create a connection for different users and databases by simply changing the environment variables.

Using environment variables with DBResourceMgr objects

You can also use this technique with a DBResourceMgr service object: simply use the variables as the values for the parameters of the ConnectDB method, as shown below:

```
self.Session = AnyDBMgr.ConnectDB(
    username = '${SITE_USERNAME}',
    password = '${SITE_PASSWORD}',
    ResourceName = '${SITE_DATABASE}');
```

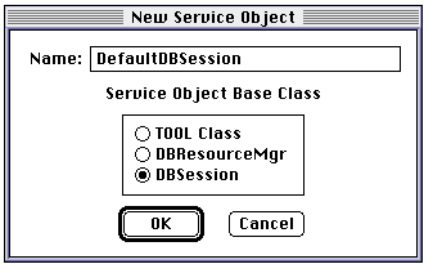
Creating a DefaultDBSession Service Object

You can establish a default database session for your SQL statements. A default database session allows you to issue a SQL statement without the **on session** clause. Using a default database session assures you that your program logic will always contain a reference to a database session. However, if you structure your code to use several sessions, you will need to change your TOOL code.

You can create a default database session for several scopes. You can create a default database session that will apply to an entire project, to a class and all of its associated methods, or to one method only. Each type is explained below.

To create a default database session for an entire project:

1. Create a DBSession service object.
2. Name it DefaultDBSession, as shown below:

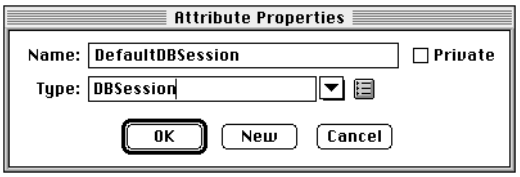


To create a default database session for a class and its associated methods:

1 Create a DBSession service object called AnotherDBSession.

2 Create a class called DataSelection.

3 Create an attribute for the above class called DefaultDBSession, as shown below.



4 In the Init method of the class, assign the DefaultDBSession attribute to the existing service object, as shown below.

```
super.init;
DefaultDBSession = AnotherDBSession;
```

5 Create a method DoSelect for the DataSelection class that executes any SQL statement.

The SQL statement will use the default database session, as will all methods of the DataSelection class.

```
begin method DoSelect
empArray : Array of employee;
sql select * into :empArray from emp;  -- no on session clause
end method;
```

To create a default database for use in a single method:

1 Assign the DBSession object to a local variable in your method:

```
etab : EmpObject = new;
DefaultDBSession : DBSession;
-- Assign to service object
DefaultDBSession = AnotherSession;
sql select * into :etab from tab;   -- no on session clause
sql insert into tab values (:etab); -- no on session clause
```

Reconnecting to a Database Session

Connections lost by a DBSession service object

The Reconnect method of the DBSession class can be used to handle connections to a database that were initiated, but lost, by a DBSession service object. For example, you can use Reconnect to resume a connection dropped due to network failure or interruption.

The Reconnect method uses the same connection information that was used to make the previous connection, whether it was coded into the application, specified in the Service Object Properties dialog, or supplied through environment variables.

The value of the IsConnected attribute must be FALSE for the Reconnect method to execute. The occurrence of a DBRemoteAccessException sets the IsConnected attribute to FALSE.

A typical use of the Reconnect method is to invoke it upon the occurrence of a DBRemoteAccessException, as in the following example:

```
when e : DBRemoteAccessException do
     defaultDBSession.Reconnect();
```

It is most efficient to catch this exception and do the reconnect in the server partition (the same partition as the DBSession service object). You must do so if a router is in use, because otherwise the reconnect could be performed on a different replicate. If a router is not in use it is still preferred, since Forte can catch the exception and perform the reconnect for one client, while other clients are unaffected by the interruption.

For more information on the Reconnect method, see the Forte online Help.

Connections lost by a DBResourceMgr service object

Lost connections are treated differently for database sessions that are created dynamically by a DBResourceMgr service object. In this case, a TOOL service object can listen for the DBRemoteAccessException, and recreate a DBSession object in the exception handler.

Disconnecting a Database Session

Once the session is no longer needed, you end the session and release its resources by invoking the Disconnect method on the DBSession object, and setting the DBSession to NIL, as shown below:

```
MySession.Disconnect();
MySession = NIL;
```

Setting the DBsession object to NIL is not required, but releases the object from memory immediately, which may be useful in more complex applications.

Vendor-Specific Notes

Informix

Informix has two mechanisms for connecting to Informix Dynamic Server databases: shared memory (local IPC) and network. A process, such as a Forte server partition, is permitted only one shared memory connection. Multiple network connections are fully supported.

Oracle

Forte uses OCI calls to communicate with Oracle. The connection is either a local Oracle connection or through SQL*Net V2.

SQL*Net V2 supports the following adapters: IPC and TCP/IP. IPC uses shared memory for local connections, and TCP uses the underlying transport protocol to connect to remote databases. Both IPC and TCP require the tnsnames.ora file.

To make a connection using SQL*Net V2:

- To make a local connection using the Bequeath adapter, use @ as a database name.
- To make a local connection using the IPC adapter, use *@TNSResourceName* as a database name.

  If you use the IPC adapter, the sqlnet.ora file must include the line:

  automatic_ipc=on
- To make a network connection using the TCP adapter, use *@TNSResourceName* as a database name.

The *TNSResourceName* points to an entry in the tnsnames.ora file that SQL*Net V2 uses to identify the requested machine on behalf of the SQL*Net client; the entry specifies the adapter and Oracle SID, among other things.

If you have difficulty connecting to Oracle, try connecting using SQL*Plus to determine whether the problem is your Oracle configuration. A SQL*Net V2 connection might look like:

```
sqlplus joe/joepass@ORACLE_COACH
```

On Unix platforms you can see the shadow Oracle process once a connection is made and the command line will show what adapter is being used.

Using Oracle with OpenVMS

Oracle on OpenVMS has further architectural limitations. Oracle 7.3 on OpenVMS no longer supports shared memory database connections. In OpenVMS, regardless of the installed version of Oracle, a Forte partition may have only one active “@” connection to Oracle. To create multiple connections, connect through the SQL*Net mailbox driver.

Rdb

You can connect to multiple Rdb databases in a single Forte application by creating as many DBSession objects as you need. However, all concurrent DBSession objects created with a single DBResourceMgr service object must use the same user name. An application that needs multiple concurrent connections with different user names must use multiple DBSession or DBResourceMgr service objects.

DBSession or DBResourceMgr connections to Rdb use the “default” Rdb alias. If an application directly issues an Rdb ATTACH statement (to create, for example, multiple Rdb database attachments in a single Rdb transaction managed by DECdtm), it should specify an explicit, non-default, alias.

Chapter 4

Working with Data Types

When you select data from a database table into a Forte application, you must be aware of the type of data you are handling. Forte converts database data to the Forte data or class type that your application specifies. For many vendor data types, the appropriate Forte types are fairly obvious, but some conversion issues are more subtle—for example, loss of precision or incompatible data types. You should choose Forte data types carefully to make sure you represent the database data accurately and consistently.

This chapter discusses using simple (scalar) TOOL data types and more complex object data types with database data. It also contains several data type conversion tables, one for each database vendor, showing allowable mappings for when you insert Forte data into a database, or select database data into Forte objects or simple data types.

For reference information about simple TOOL data types, refer to the *TOOL Reference Manual*; for details about the object data types, refer to the online Help.

Using Database Data with Forte

When you use database data in a Forte application, one choice you have is whether to bring database data in as a simple scalar data type or as a class type. You also must indicate which scalar or class type is appropriate for each database value. Generally this means mapping database column data types to the appropriate Forte data types. While some mappings are obvious and natural, there are more subtle considerations that you may very well encounter, including representing money, dates and times, internationalization issues, and numeric precision. This chapter addresses many of these issues; you may also wish to refer to the Forte online Help or the *Forte 4GL Programming Guide*.

Transferring data

In Forte applications, you may store data in variables or in the attributes of objects, which are either scalar types or predefined subclasses of the Forte DataValue class. In addition, some TOOL SQL extensions to some of the SQL statements allow for an automatic mapping of database result values to user-defined object attributes, and a mapping of user-defined object attributes to columns that will be inserted into a database.

Using Simple Data Types in TOOL

Simple TOOL data types are appropriate and easy to use when you transfer data between a database and a Forte application and you do not encounter NULL values; for example, to retrieve integer data, you would simply declare a variable of type integer, and perform a select statement into that variable, as shown below:

```
year : integer;
sql select YearPainted into :year from Paintings
where Title = 'Mona Lisa'
on session MySession;
```

Using Nullable DataValue Subclasses

The Framework library contains several classes for data storage and manipulation, including subclasses to represent NULL values from database tables. The nullable classes are useful because database tables often contain NULL data. The following table lists the DataValue classes and corresponding nullable subclasses:

Class Data Type	Nullable Variant
BinaryData	BinaryNullable
BooleanData	BooleanNullable
DateTimeData	DateTimeNullable
DecimalData	DecimalNullable
DoubleData	DoubleNullable
ImageData	ImageNullable
IntegerData	IntegerNullable
IntervalData	IntervalNullable
TextData	TextNullable

You use these classes as you would any other Forte class: define an attribute or variable of the class type you need, and then construct the object before you use it. For example:

Example: using a DataValue class

```
Comments : TextData;
Comments = new;
```

Data Type Conversion

The next several sections contain tables that show the internal data types for DB2, Informix, ODBC, Oracle, Rdb, and Sybase databases, with corresponding TOOL data types. Note that the direction the data flows affects the appropriate data conversion. That is: certain TOOL data types can accept certain internal database types (for example, from a **sql select** statement), while the internal database types may accept other TOOL types (from a **sql insert** statement).

Handling NULL values

Remember that if you select a NULL value into a scalar variable, scalar attribute, or non-nullable object, you will get an error of the DatatypeException class. To avoid this, use the nullable variant of the DataValue subclass (for example, TextNullable).

Overflow

If you use a type that is too small, such as i1 for a value bigger than 256, you may get conversion errors at runtime.

Reading the Data Type Conversion Tables

In the following data conversion tables, the first entry for each category represents the most "natural" mapping of the database type to Forte TOOL type. We have also grouped several related data types into one representative, generic TOOL data type. These groupings are shown in Table 2.

Table 2 *Legend for the Data Type Conversion Tables*

Generic Data Type	Represents These Types
Boolean	boolean, BooleanData, BooleanNullable
Datetime	DateTimeData, DateTimeNullable
Integer	integer, long, short, i4, ui4, i2, ui2, i1, ui1, IntegerData, IntegerNullable
Interval	IntervalData, IntervalNullable
Text	String, TextData, TextNullable
Double	double, float, DoubleData, DoubleNullable, DecimalData, DecimalNullable
Binary	BinaryData, BinaryNullable, ImageData

For example, if an Oracle VARCHAR2 can be selected into a generic data type called "Text" it can also be selected into a String or objects of type TextData or TextNullable.

Specific notes on data conversion

The following additional notes may apply to one or more of the data conversion tables.

(a) Can be used as a host TOOL variable only if the selected column's data contains appropriate values for the TOOL data type (for example, a VARCHAR column can be selected into an IntegerData TOOL variable only if the column's value consists of numeric characters).

(b) The database data type can contain values that are larger than can be represented in the TOOL variable data type. Selecting out-of-range values may result in undetected arithmetic overflow/underflow conditions (for example, a BIGINT can be selected into an IntegerData TOOL variable, but column values exceeding 2,147,483,647 in absolute value cannot be represented in an IntegerData object and will be incorrectly stored). Use these TOOL data types only if you are certain that no out-of-range values will be stored in the database. Note that you may select these columns into a "large" TOOL data type (DoubleData, for example) and then assign it to another TOOL data type (for example, IntegerData) if its value is within IntegerData's range.

(c) When selecting a database column value that is "scaled" (for example, either a scaled integer or a floating point data type) into an exact numeric TOOL data type (IntegerData, i2, and so on), the fractional part of the value (the part to the right of the decimal point) will be lost.

(d) When inserting a TOOL data value into the database, the TOOL value must be appropriate for the database data type. For example, a BinaryData object inserted into a VARCHAR column cannot contain any embedded binary zeros, and a TextData object inserted into a TINYINT column must contain numeric characters and the total value must be within the -256 to +255 range for TINYINT.

(e) When inserting from a TOOL data type that has a larger range than the database data type (for example, from a TOOL DoubleData into a Rdb SMALLINT), overflow/underflow conditions may result. These may be undetected. Be careful when inserting from TOOL data types that are “larger” than their corresponding database data types. The Forte application should ensure that out-of-range values are not inserted into the database.

(f) (Rdb and DB2 only) Rdb's DATE ANSI and DB2’s DATE data types contain no hour/minute/second component. If the TOOL DateTimeData value contained such a value, that part will be lost. Use the Rdb or DB2 TIMESTAMP data type when values will include both year/month/day and hour/minute/second components.

(g) (Rdb and DB2 only) Rdb's and DB2’s TIME data type contain no year/month/day component. If the TOOL IntervalData value contains a year/month/day component, it will be lost. Use the Rdb or DB2 TIMESTAMP data type when values will include both year/month/day and hour/minute/second components.

(h) (DB2 and ODBC only) Limitations in the DB2 CLI V2 interface prevent data type conversion for parameter markers. Parameter marker TOOL variables must therefore be of the data class that directly maps to the data type of the DB2 column or parameter marker. When using the DBSession classes directly (in lieu of TOOL SQL), it is possible to force data type conversion by explicitly setting the input marker’s DBColumnDesc.DBDataType attribute to the value associated with the DB2 column’s actual data type. For example, to insert an IntervalData object into a DB2 VARCHAR column, the input DataSet’s associated DBColumnDesc.DBDataType attribute must be set to DB_DT_CHARACTERVARYING before the Execute method is invoked.

DB2 Data Conversion Table and Notes

Using DATE

In this release, TOOL parameter markers cannot be used for DB2 DATE data type columns. DateTimeData objects are assumed to be associated with DB2 TIMESTAMP columns. As a workaround, applications can affect DATE columns only by building queries using the DBSession classes directly. Specifically, when input values are set in the inputDataSet (returned by the Prepare method), the associated DBColumnDesc.DBDataType attribute must be changed to the value DB_DT_ANSIDATE. The Execute method will then be able to bind the parameter to the DB2 DATE column.

Using TIMESTAMP

DB2's TIMESTAMP data type can contain fractional seconds to the resolution of a microsecond. Forte DateTimeData data class, however, is limited to millisecond resolution. When DB2 TIMESTAMP columns are selected into Forte DateTimeData objects, the fractional second component is truncated to millisecond resolution.

See notes (a) through (h) and Table 2 on page 57 as you use this table.

DB2 Data Types	can be selected into these generic TOOL types	and inserted from these TOOL types
VARCHAR LONGVARCHAR CLOB	TEXT Integer (a) Boolean (a) DateTime (a) Double (a) Interval (a) Binary	TEXT (h)
CHAR	TEXT Integer (a) Boolean (a) DateTime (a) Double (a) Interval (a) Binary	TEXT (h)
GRAPHIC VARGRAPHIC LONGVARGRAPHIC	TEXT Binary	TEXT (h)
SMALLINT	INTEGER Double Text	INTEGER (e, h)
INTEGER DECIMAL NUMERIC	INTEGER Double Text	INTEGER (h)
DECIMAL FLOAT	DOUBLE Integer (b c) Text	DOUBLE (h)
DATE	DATETIME Text	DATETIME (f) (see note above)
TIMESTAMP	DATETIME Text	DATETIME (h) (see note above)
TIME	INTERVAL Text	INTERVAL (g, h)
BLOB	BINARY Text	BINARY

Informix Data Conversion Table and Notes

Using DATETIME

The Informix DATETIME datatype is actually a family of 28 data types. Along with the DATE datatype, these are mapped to the Forte DateTimeData class. When used in an SQL **where** clause, the **set** clause of an **update**, or as an input parameter to an Informix database procedure, only Informix DATETIME YEAR TO SECOND database columns and parameters can be referenced by a DateTimeData host variable. To reference columns of other DATETIME subtypes in these cases, use a TextData or string and specify the precise Informix string representation for the column or parameter.

Using INTERVAL

The Informix INTERVAL data type is actually a family of 18 data types. These are mapped to the Forte IntervalData class. When used in an SQL WHERE clause, the **set** clause of an **update**, or as an input parameter to an Informix database procedure, Forte maps IntervalData values containing a year or month to an Informix INTERVAL YEAR TO MONTH value and other Interval Data values to Informix INTERVAL DAY TO FRACTION values. To reference other types of INTERVAL columns or parameters, use a TextData or string host variable and specify the precise Informix string representation for the column or parameter.

See notes (a) through (h) and Table 2 on page 57 as you use this table.

Informix Data Type	can be selected into these generic TOOL types	and inserted from these TOOL types
CHARACTER CHAR CHARACTER VARYING NCHAR NVARCHAR TEXT VARCHAR	TEXT Integer (a) Double (a) DateTime (a) Interval (a)	TEXT Integer (d) Double (d) Binary (d) DateTime (d) Boolean
INTEGER INT SERIAL SMALLINT	INTEGER Double Binary Boolean (a) Text	INTEGER Double (e) Text (d)
DEC DECIMAL DOUBLE PRECISION FLOAT MONEY NUMERIC REAL SMALLFLOAT	DOUBLE Integer (c) Text Boolean (a)	DOUBLE (e) Integer (e) Text (d,e)
BYTE	BINARY Text (a)	BINARY Text
DATE DATETIME	DATETIME Text	DATETIME Text (d) (see note above)
INTERVAL	INTERVAL	INTERVAL (see note above)

ODBC Data Conversion Table and Notes

Only ODBC data types may be used with a Forte ODBC resource manager. The mapping of database to ODBC data types is database-specific; consult your database driver documentation to obtain the mappings you require.

If you insert a data type of BIGINT with a fractional component into generic data type DOUBLE, it will be truncated (no rounding).

Data conversion for ODBC data sources

This manual does not include data type mappings for data sources such as SQL Server to ODBC, as those mappings are supported and updated by other vendors. For example, to see Microsoft SQL Server to ODBC datatype mappings you are referred to the documentation for Microsoft SQL Server.

See notes (a) through (h) and Table 2 on page 57 as you use this table.

ODBC Data Type	can be selected into these generic TOOL types	and inserted from these TOOL types
VARCHAR LONGVARCHAR	TEXT Integer (a) Boolean (a) DateTime (a) Double (a) Interval (a) Binary	TEXT (h)
BINARY VARBINARY LONGVARBINARY	BINARY Text DateTime	BINARY
CHAR	TEXT Integer (a) Boolean (a) DateTime (a) Double (a) Interval (a) Binary	TEXT (h)
BIT	INTEGER Boolean Double	INTEGER Boolean Text (d)
SMALLINT	INTEGER Double Text	INTEGER (e) Double (e) Text (d)
TINYINT	INTEGER Double Text	INTEGER (e) Double (e) Text(d)
BIGINT	DOUBLE Integer (b) Text	DOUBLE (see note above) Integer Text (d)
INTEGER DECIMAL NUMERIC	INTEGER Double Text	INTEGER Double Text (d)
DECIMAL FLOAT	DOUBLE Integer (b,c) Text	DOUBLE Integer Text (d)
DATE	DATETIME Text	DATETIME Text (d)
TIMESTAMP	BINARY	BINARY
TIME	DATETIME Text	DATETIME Text (d)

Oracle Data Conversion Table and Notes

An Oracle number type is integer if scale is 0, floating point otherwise.

See notes (a) through (h) and Table 2 on page 57 as you use this table.

Oracle Data Type	can be selected into these generic TOOL types	and inserted from these TOOL types
VARCHAR2 CHAR LONG	TEXT Integer (a) Double (a) DateTime (a) Interval (a)	TEXT Integer (d) Double (d) Binary (d) DateTime (d) Boolean
NUMBER	INTEGER (b) Double (see note above) Binary Boolean (a) Text	INTEGER Double (e) Text (d)
ROW ID	TEXT	Can't insert into ROWID
RAW MLS LABEL MLS LABEL	Not supported	Not supported
LONG RAW RAW	BINARY Text	BINARY Text
DATE	DATETIME Text	DATETIME Text (d)

Rdb Data Conversion Table and Notes

Using DOUBLE PRECISION and DoubleData

On the VAX only, Rdb's DOUBLE PRECISION columns are stored in "G_FLOATING" format, while TOOL's DoubleData data type is stored in "D_FLOATING" format. G_FLOATING's range is larger than D_FLOATING (see Digital's *VAX Architecture Handbook* for details), thus, very large column values (with exponents exceeding 38 in absolute value) cannot be stored in a TOOL data class. These cases are best handled by using CAST and/or arithmetic functions on the select statement to reduce the column value to a storable range. Note that on Alpha AXP OpenVMS, both Rdb DOUBLE PRECISION and TOOL DoubleData data types are stored in "G_FLOATING" format and are entirely compatible.

Inserting scaled data

When inserting a TOOL data value that is "scaled" (either a DecimalData or DoubleData data type, for example) into an exact numeric Rdb column (TINYINT, SMALLINT, INTEGER, or BIGINT), the fractional part of the value (the part to the right of the decimal point) will be lost.

Using LIST OF BYTE VARYING

When inserting into a LIST OF BYTE VARYING column, the byte string is written in segments of the maximum segment size defined when the column was originally created. When these columns are fetched, all the columns' segments are concatenated to form the BinaryData object. Note that Rdb SQL does not permit LIST OF BYTE VARYING columns to be updated; they may only be inserted and selected.

Using DBKEYS

DBKEYS cannot be inserted into a table, but can be used as host variables in a WHERE clause to identify a row to be updated, deleted, or fetched.

See notes (a) through (h) and Table 2 on page 57 as you use this table.

Rdb Data Type	can be selected into these generic TOOL types	and inserted from these TOOL types
VARCHAR	TEXT Integer (a) Boolean (a) DateTime (a) Double (a) Interval (a) Binary	TEXT Integer Boolean DateTime Double Interval Binary (d)
CHAR	TEXT Integer (a) Boolean (a) DateTime (a) Double (a) Interval (a) Binary	TEXT Integer (e) Boolean (e) DateTime (e) Double (e) Interval (e) Binary (d,e)
TINYINT (unscaled)	INTEGER Double Text	INTEGER (d,e) Double (d,e) Text (d,e)
TINYINT (scaled)	DOUBLE Integer (c) Text	DOUBLE (d,e) Integer (d,e) Text (d,e)
SMALLINT (unscaled)	INTEGER Double Text	INTEGER (d,e) Double Text (de) (see note above)
SMALLINT (scaled)	DOUBLE Integer (c) Text	DOUBLE (e) Integer (e) Text (d,e)

Rdb Data Type	can be selected into these generic TOOL types	and inserted from these TOOL types
INTEGER (unscaled)	INTEGER Double Text	INTEGER Double (e) Text (d,e) (see note above)
INTEGER (scaled)	DOUBLE Integer (c) Text	DOUBLE (e) Integer (e) Text (d,e)
BIGINT (unscaled)	DOUBLE Integer (b) Text	DOUBLE Integer Text (d,e)
BIGINT (scaled)	DOUBLE Integer (b,c) Text	DOUBLE (e) Integer Text (d,e)
REAL	DOUBLE Integer (b c) Text	DOUBLE Integer Text (d)
DOUBLE PRECISION	DOUBLE (see note above) Integer (b,c) Text	DOUBLE Integer Text (d)
DATE VMS	DATETIME Text	DATETIME Text (d)
DATE ANSI	DATETIME Text	DATETIME (f) Text (d,f)
TIMESTAMP	DATETIME Text	DATETIME Text (d)
TIME	INTERVAL Text	INTERVAL (g) Text (d,g)
INTERVAL	INTERVAL Text	INTERVAL Text (d)
LIST OF BYTE VARYING (see note above)	BINARY Text (a)	BINARY Text (see note above)
DBKEY	BINARY	BINARY

Sybase Data Conversion Table and Notes

See notes (a) through (h) and Table 2 on page 57 as you use this table.

Sybase Data Type	can be selected into these generic TOOL types	and inserted from these TOOL types
VARCHAR CHAR	TEXT Integer (a) Boolean (a) Double (a) DateTime (a) Binary Interval (a)	TEXT Binary (d)
TINYINT SMALLINT INT	INTEGER Double Text Boolean	INTEGER (d) Double (e) Boolean
REAL FLOAT MONEY SMALLMONEY	DOUBLE Integer (b) Text Boolean	DOUBLE Integer Boolean
DATETIME SMALLDATETIME	DATETIME Text	DATETIME Text
VARBINARY BINARY	BINARY Text DateTime	BINARY Integer Boolean
BIT	INTEGER Text Boolean	INTEGER Boolean
TEXT	TEXT Binary	TEXT Binary DateTime
IMAGE	BINARY Text	BINARY Text DateTime

Chapter 5

Manipulating Data

The Forte database interface allows application programmers to write applications that use the full potential of relational databases. This chapter describes how to add SQL statements to a Forte application so that it can access and manipulate database data.

You can use TOOL SQL statements to query or update data. You can also use the Forte database management classes found in the GenericDBMS library to access and manipulate database data.

This chapter describes the various ways you can query, insert, update and delete database data. It includes several examples, some drawn from the Forte examples. It also describes creating and naming the Forte objects into which you will select data. At the end of the chapter are database vendor-specific notes.

For reference information about TOOL SQL statements, refer to Appendix B, "TOOL SQL Statement Reference."

Accessing Database Data from Forte

While Forte provides full support for building SQL statements into a Forte application, programmers should keep in mind some general questions when working with database data. These questions include:

- If I use non-standard SQL (that is, vendor-proprietary extensions to SQL) in my application, will that prevent my application from working with any database that it must work with?
- When should I use TOOL SQL statements or the classes in the GenericDBMS library?
- When moving data between a database and Forte application, how can I assure that I use compatible data types?

Standard ANSI SQL

Forte passes all SQL to the database including vendor-specific extensions. Using only standard ANSI SQL increases the likelihood that your applications will run on any of the database management systems that Forte supports. Note, however, that Forte does not do any translation of TOOL SQL before passing it to the DBMS; that is, compiling a TOOL method containing SQL does not guarantee that it will execute on any database. If you need to write an application that is portable across multiple databases, you must ensure that you use a subset of SQL that works for all of those particular databases.

A small number of vendor-specific SQL extensions are not supported; to see a list, refer to "Unsupported Database Features" on page 22.

When to use TOOL SQL or database classes

To manipulate database data, you can use either TOOL SQL statements or Forte GenericDBMS classes.

TOOL SQL statements are more convenient to use, particularly when you can embed specific SQL statements in your application. An example is a query for which you know the selection criteria and the names of the columns to be returned, even though you might not know what the selection criteria *values* are. You can embed the known column and table names, and use variables to get a selection value from the user (such as, generate the report for Department "10").

However, TOOL SQL cannot be used to write SQL statements that are generated at runtime. An application that allows SQL statements to be generated and executed "on-the-fly" has the potential to be a more flexible, multi-purpose application. To write such an application, you use the classes in the Forte GenericDBMS library, particularly the DBSession class. The section "Using Forte Classes to Execute SQL" on page 85 begins the discussion of using the DBSession class interface to write these applications.

Application performance is the same, whether you use TOOL SQL or methods on the DBSession class, because TOOL statements are translated into invocations of methods on the DBSession class at compile time.

Transferring data

In Forte applications, you may store data in variables or in object attributes, which are either scalar types or predefined subclasses of the Forte DataValue class. In addition, TOOL SQL extensions to some of the SQL statements allow for an automatic mapping of database result values to user-defined object attributes, and a mapping of user-defined object attributes to columns that will be inserted into a database.

You must also consider issues of data type conversion when integrating database data into your Forte application. While examples in this chapter illustrate some data type conversion, you should also refer to Chapter 4, "Working with Data Types" to see how Forte converts and uses the data types for your particular database vendor.

Using Forte Names in SQL Statements

When you use Forte names (variables, attributes, and so on) in TOOL SQL statements, you must preface the Forte names with colons to distinguish them from database names (columns and tables). For example:

```
name : TextData = new;
year_born : integer = 1900;
sql select ptr_name into :name from painter_table
  where birth < :year_born;
```

Note

You do not precede Forte names with colons in two TOOL SQL statements (**sql open cursor** and **sql execute procedure**). These statements both take a parameter list and a syntax error occurs if a colon precedes a parameter value.

TOOL SQL Statements

The following table lists the TOOL statements that correspond to SQL DML statements used to query and update database data. You can find reference information for these statements in Appendix B, "TOOL SQL Statement Reference."

TOOL SQL Statement	Description
sql select	Retrieves rows from a database table.
sql delete	Removes rows from a database table.
sql insert	Adds a new row to a database table.
sql update	Replaces values in a database table.
sql open cursor	Opens a cursor.
sql close cursor	Closes a cursor.
sql fetch cursor	Retrieves rows from a cursor.
sql execute immediate	Executes a single SQL statement specified as a literal string, a string variable, or a TextData variable.
sql execute procedure	Executes a database procedure.

Use **on session** clause to identify a database session

For a TOOL SQL statement to execute in a database, it must be associated with a connected database session. You can initiate a database connection in three ways, described in "Connecting to a Database" on page 38. To associate a TOOL SQL statement with a database session, you use the **on session** clause with the name of the DBSession object that you wish to use for the connection. Note that not all statements require or allow the **on session clause**.

For example, the following query retrieves a single row from the database, using the session specified by the MySession DBSession object, into an object referenced by a variable called emp:

```
sql select * into :emp from emptable
where name = 'Smith'
  on session MySession;
```

Using Conditional TOOL for Vendor-Specific Code

In your SQL statements, you can use any vendor-specific extensions to the ANSI SQL syntax that are allowed by the particular database management system you are using. Examples of extensions to ANSI standard SQL include the Rdb cast statement, Sybase compute statement, or Oracle outer join.

However, if you include a vendor-specific clause in a SQL statement, your code is no longer generic, and will work only for database products that support that particular syntax. Also note that because Forte passes vendor-specific syntax directly to the database, it does not detect syntax errors within vendor-specific clauses.

If your application must execute against multiple databases and you want to use vendor-specific SQL extensions, then you can use conditional TOOL code. Conditional code allows you to issue different SQL statements depending on which type of database an application is connected to.

DBVendorType attribute contains the database vendor

To write conditional code that is based on the database vendor, you use the DBVendorType attribute of the DBSession class. This attribute contains the database vendor for the current database session. By using the DBVendorType attribute in a **case** or **if** statement, you can include vendor-specific SQL extensions that will be invoked only for the databases that support them.

The following example shows the use of DBVendorType to issue vendor-specific SQL:

```
case DefaultDBSession.DBVendorType
when DB_VT_ORACLE do
  sql execute immediate
    'create table emp (id int, hired date, salary float)';

when DB_VT_INFORMIX do
  sql execute immediate
    'create table emp (id integer, hired datetime year to fraction
(3),
    salary money)';

when DB_VT_SYBASE do
    sql execute immediate
     'create table emp(id int, hired dateTime, salary float)';

when DB_VT_DB2 do
  sql execute immediate
     'create table emp (id integer, hired timestamp, salary
       float)';

when DB_VT_Rdb do
  sql execute immediate
     'create table emp (id integer, hired timestamp, salary
       double precision)';
end case;
```

Using TOOL Statements to Query Data

The following sections describe various ways of retrieving data from a database, with several examples. Also note the first few sections that describe how to create and name objects and variables into which you will select database data.

Selecting Data and Object Creation

When you query data (using either the **sql select** or cursor statements) you may assign the retrieved data to a Forte object. In some cases, Forte creates the object for you; in other cases you must explicitly create the objects yourself.

In a simple select into a set of DataValue objects, you must create the DataValue objects before the select, as in the following:

Example: create the DataValue object before the select

```
i : IntegerNullable = new;
f : DoubleNullable = new;

sql select intval, floatval into :i, :f
  from myTab on session MySession;
```

You must also create objects that are used as attributes to the object. For example:

```
class myClass
  has public i : IntegerNullable;
  has public f : DoubleNullable;
end class

method init
  begin
    super.Init;
    self.i = new;
    self.f = new;
  end method;

method doSelect
  begin
    myObj : myClass = new;
    sql select * from myTab into :myObj on session MySession;
```

Selecting into arrays

You do not need to create an instance of the array when you select into an array of objects. Forte also creates the rows of the array during the select. For example:

```
myArray : Array of myClass;
sql select * from myTab into :myArray on session MySession;
```

Forte automatically creates the objects on each interation in a **for** loop, as shown below:

```
for (myObj : myClass) in sql select * from myTab
  on session mySession do
... myObj is created for you...
end for;
```

You must explicitly create the objects before a **sql fetch cursor** statement, as shown below:

```
-- Assume cursor myCursor with the following definition
-- select * from myTab on session mySession

myCursor_ref : myCursor;
sql open cursor myCursor_ref on session mySession;
while (TRUE) do
  i : integer;
  i : IntegerNullable = new;
  f : DoubleNullable = new;

  i = sql fetch cursor myCursor_ref into :i, :f;
end while;
```

Selecting a Single Row

You can select a single row from a database into a Forte variable or object attribute using the TOOL **sql select** statement. If more than one row would be returned, Forte raises an exception of the DBOperationException class, and does not return any data. To select more than one row, you must select into an array or a cursor, or use a **for** loop—these techniques are explained below.

Selecting into a Variable

You can select data from a database into one or more Forte variables. In the following example, note that the two Forte variable names are preceded with colons when used in the TOOL statement:

```
name : string = 'Picasso';
t_comments : TextData = new;
t_country : string;
sql select comments, country into :t_comments, :t_country
from ArtistTab
where Name = :name
on session MySession;
```

Selecting into an Object

You can use the **sql select** statement to retrieve one row from a database table and store the row values in a Forte object. To do so, the names of the object attributes must match the names of the retrieved columns. (Attribute names are case-insensitive, while column names may be case-sensitive depending upon the database vendor.) You specify the conditions that the row must meet, as in the following example:

```
painter : Artist = new;
name : TextData = new(value='De%');
sql select * into :painter
    from ArtistTab
    where Name like :name
    on session MySession;
```

When Attribute and Column Names Match

Selecting all columns

When a **sql select** statement selects all columns from a table into an object (for example, using select *), Forte retrieves all table columns. Every column with a matching attribute is selected into that attribute. That is, if the column name and attribute name match (case-insensitive), the attribute receives the column's value.

Number of columns and attributes differs

If the table has columns with no corresponding attribute, those columns are ignored. Similarly, if the object has attributes with no corresponding columns, those attributes are untouched. Note that if the number of attributes is significantly higher than the number of columns, you will get better performance if you explicitly specify the column names in the select statement.

Selecting a subset of columns

If a **sql select** statement selects specific columns from a table, those columns must have matching attributes in the receiving object. For example, if you select the name and birthdate from ArtistTab, then the painter object must also have attributes named name and birthdate.

When Attribute and Column Names Do Not Match

When you need to select into an object whose attribute names do not match the column names of the source database table, you can do so in several ways:

You can "rename" the columns in the select list of the statement, as in:

```
sql select col1 "attr1", col2 "attr2", into :obj from tablea...;
```

You can select directly into the attributes, as in:

```
sql select col1, col2 into :obj.attr1, :obj.attr2 from tableA...:
```

You can also use the TOOL **for** statement to achieve column renaming in order to match column and attribute names, as described in the next section.

When Attributes are of Other Class Types

In most cases, if there are any attributes in the object that have user-defined class types (other than DataValue types), Forte ignores these attributes even when they have matching names. To select into such an object, you must rename the column in the select list to reference the class to which the attribute belongs. For example, assume you have the two classes AddrClass and EmpClass:

AddrClass

```
class AddrClass inherits from Object
has public
  Street : TextData:
  State : TextData;
  Zip : Integer;
...
```

EmpClass

```
Class EmpClass inherits from Object
has public
Name : TextData;
Address : AddrClass;
```

Also assume the Emp table has the following definition:

```
table Emp
(ID int, Name char(20), Street char(30), City char(20), Zip int)
```

To select into an EmpClass object, which contains an attribute Address of type AddrClass, you must rename the columns that reference the Street, City, and Zip attributes, as shown below:

Example: Renaming columns

```
empObj: EmpClass;
sql select
  ID,
  Name,
  Street      "Address.Street",
  City        "Address.City",
  Zip         "Address.Zip"
  into :empObj
  from Emp on session MySession;
```

Vendor restrictions on column renaming

Note that column renaming may not be portable across all databases and is subject to restrictions of each database vendor. Some of the known vendor-specific restrictions follow:

- For Rdb, columns are renamed using the ANSI SQL “as” clause.
- For Oracle, the length of a column alias is limited to 32 characters.
- DB2 does not allow you to rename a result column arbitrarily (eliminating the above approach).
- Informix does not allow an embedded period when renaming a column (also eliminating the above approach).

If the syntax in the example above does not work for your database, you can use a **for** loop to achieve the same effect; see the *TOOL Reference Manual* for more information on the **for** statement.

When Attributes are Inherited

As is true with all TOOL objects, you can select data into an object that is a subclass of another object, using the attributes of both classes. For example, say you have a superclass Person with attributes Name, Address, and Age. Person has a subclass called Employee, with the attribute Salary. Your database table has columns Name, Address, Age, and Salary. You can successfully perform the following select statement:

Example: selecting into objects with inherited attributes

```
info : Employee = new;
sql select Name, Address, Age, Salary into :info
  from empTable on session MySession;
```

Selecting Multiple Rows into Arrays

To select data into an array, you declare an array object and select into it (you do not need to instantiate it). Also, you do not need to add rows to the array—when you perform the select, the array will accommodate as many rows as are retrieved. Any rows that existed in the array before the select will be gone.

You can use the **into** clause to select multiple rows into an array, as shown in the following example:

Example: selecting multiple rows Into an array

```
painter.ListOfPaintings : Array of Painting;
sql select * into :painter.ListOfPaintings from ArtistTab
    where Painter = :painter.name
    on session MySession;
```

Selecting Multiple Rows using the TOOL for Statement

You can use the **for** statement with a **sql select** statement to retrieve multiple rows from a database.

Example: for statement

```
painters : Array of Artist;
for (a : Artist) in sql select * from artistTab
on session MySession do
-- A new Artist object is allocated each time through
    painters.AppendRow(a);
end for;
```

Selecting Multiple Rows using Cursors

A cursor is a row marker that you use to work with a set of rows from a database. Cursors have the following benefits:

- Cursors are useful if you do not know how many rows a query will retrieve—you can simply fetch one or more rows at a time from the cursor's result set.
- A cursor is defined in one place only—the Cursor Workshop. You can use a cursor in multiple methods, but if you need to modify the cursor's select statement, you need only make the change once in the Cursor Workshop.

You can use a cursor in several different ways:

- You can use the **sql open cursor, sql fetch cursor**, and **sql close cursor** statements to step through the cursor's result set one row at a time.
- You can use the same statements to fetch the entire result set into an array.
- You can use the same statements, but with **sql fetch next *n*,** to step through the result set *n* rows at a time.
- You can use the **for** statement to repeat a statement block for each row in the result set of the cursor.

For Oracle, Rdb, and Informix, you can also use a cursor with the **sql update ... where current of** or **sql delete ... where current of** statements to update or delete the row to which the cursor is pointing (a "positioned update"). When you define the cursor, you need to include the for update clause in the cursor's SQL statement. See "SQL Update" on page 145 and "SQL Delete" on page 128 for more information about using positioned updates.

Defining a Cursor

You define a cursor in the Cursor Workshop (described in *A Guide to the Forte 4GL Workshops*). A cursor definition includes a cursor name and an underlying query. Because a cursor definition is associated with a Forte project, you can refer to the cursor in any method in the project.

An example of a cursor definition follows:

```
cursor BlobCursor(name: Framework.string)
begin
  select BlobValue from ArtistBlob
  where Name like :name;
end;
```

As described in the *TOOL Reference Manual*, most TOOL statements use a cursor reference rather than the cursor's name. (The TOOL **for** statement uses the cursor name rather than a cursor reference.) In the following code fragment blobCurs is the cursor reference for the cursor named BlobCursor:

Example: Using a cursor reference

```
blobCurs : BlobCursor;
begin transaction
     sql open cursor blobCurs(name) on session self.MGRSession;
     rowcount =  (sql fetch cursor blobCurs into :binData);
     sql close cursor blobCurs;
. . .
```

Retrieving Rows

To retrieve rows from a cursor, you must first open the cursor with the **sql open cursor** statement. You can then use the **fetch** statement to retrieve one or more rows at a time. Finish by using the **sql close cursor** to close the cursor. For best performance, you should enclose all statements relating to one use of a cursor in an explicit Forte transaction.

sql open cursor statement

The **sql open cursor** statement executes the **select** statement associated with the cursor and positions the cursor before the first row in the result set. At this point you specify the values for any placeholders used in the original cursor declaration. (Placeholders are mechanisms through which you can pass data into your methods at runtime.):

```
dbcursor : empcursor;
empid : integer;
sql open cursor dbcursor (empid) on session MySession;
```

In this example, the cursor name is empcursor and the cursor reference is dbcursor.

sql fetch cursor statement

Use the **sql fetch cursor** statement to move one or more rows at a time through the result set. This statement need not be in the same method as the **sql open cursor** statement but it should be in the same explicit transaction. The first time you use the **sql fetch cursor** statement, Forte moves the cursor to the first row (or set of rows) in the result set and retrieves the data into the specified TOOL variables. With each successive **sql fetch** statement, Forte moves the cursor forward one or more rows, retrieving the data into the specified TOOL variables. You continue using **sql fetch cursor** to move through the result set until you reach the last row.

```
emp : employee = new;
sql fetch cursor dbcursor into :emp;
```

sql close cursor statement

Use the **sql close cursor** statement to close the cursor. After the cursor is closed, it cannot be used again until you give another **sql open cursor** statement to open it.

```
sql close cursor dbcursor;
```

Fetching into an Array

You can also fetch the entire result set into an array. For example:

```
empArray : array of employee;
sql fetch cursor dbcursor into :empArray on session MySession;
```

Fetching an Arbitrary Number of Rows

Rather than having to fetch only one row or all rows, you can use the **sql fetch cursor** statement with the **next** key word to specify the maximum number of rows to fetch into an array. For example:

```
dbsess  :  DBSession;
rows    : integer = 1;
dbcursor :  artist_cursor;
painters :  array of artist  = new;

sql open cursor dbcursor ('%') ON SESSION dbsess;
while (rows > 0) do
    sql fetch next 10 from cursor dbcursor into :painters;
    rows = painters.Items;
    -- do something with painters ...
end while;
```

Repeating a Statement Block

You can use a **for** statement to create a loop that will fetch and process one record at a time. The **for** statement automatically opens the cursor, fetches the rows one at a time as it goes through the loop, and then closes the cursor. To use the **for** statement with a cursor, you use the cursor name rather than a cursor reference, as follows:

```
MySession : DBSession =  ...get a session...
for (emp : employee)
  in cursor empcursor(empid) on session MySession do
    -- A new emp object is allocated each time through
    -- user code
end for;
```

Note that this operation is identical to using the **for** statement with a **sql select** statement (described above). Using a cursor in the **for** statement is especially useful when the cursor is also used in other parts of the application.

Using TOOL to Update Data

The next several sections discuss using TOOL SQL statements to insert, update, and delete database data.

Inserting a Single Row

You use a **sql insert** statement to insert data directly into a table.

Inserting Variables

You can insert data from variables using the **sql insert** statement. For example:

```
t_comments : TextData = new(value = 'Comments');
t_name : string = 'Picasso';
t_country : string = 'Spain';
sql insert into ArtistTab
   (name, comments, country)
  values (:t_name, :t_comments, :t_country)
  on session MySession;
```

Inserting from an Object

Use the **sql insert** statement to add a new row to a table from an object. If you do not specify a column list, Forte uses the object's attributes as the column list. If you do not specify a column list and the database uses case-sensitive column names (for example, Sybase or Microsoft SQL Server), the attribute names must exactly match the database column names (that is, both spelling and case must match).

```
NewPainting : Artist = new;
sql insert into ArtistTab
  values (:Newpainting)
  on session MySession;
```

You can also insert into a subset of the columns by specifying a column list:

```
NewPainting : Artist = new;
sql insert into ArtistTab(name, comments)
  values (:newPainting)
  on session MySession;
```

Note If you do not instantiate an object (using "new"), it is treated as a NULL by the database.

Inserting Multiple Rows

To insert a set of rows into a table, you can use a **sql insert** statement with an array in the **values** clause. This is the most efficient way to insert a set of rows. For example:

```
artists :  Array of Artist = new;
-- Fill in array...
sql insert into ArtistTab values (:artists) on session MySession;
```

Updating a Row

To replace the current column values in selected rows, use the **sql update** statement. For example:

```
sql update ArtistTab set born = :birthyear
  where name = :vname on session MySession;
```

Unless you specify a **where** clause, values in all rows are updated.

Deleting a Row

To remove one or more rows from a table, use the **sql delete** statement. For example:

```
sql delete from ArtistTab where born < 1500;
```

If you do not include a **where** clause, all rows in the table are deleted.

Executing a Single SQL Statement

Use the **sql execute immediate** statement to execute a SQL statement that you enter directly as a literal string, or a statement that is stored in a Forte variable or attribute. You use the **sql execute immediate** statement to execute SQL statements that are not explicitly part of TOOL, or to execute statements constructed at runtime. For example, you can use it to issue DDL (data definition language) statements, such as the following:

```
sql execute immediate 'create table ArtistTab(
  Name char(30) not null, Country char(30) not null,
  Comments varchar(200) not null)'
  on session MySession;
sql execute immediate
   'grant all on ArtistTab to public' on session MySession
```

See DynamicDataAccess example

Project: QueryMgr • **Class**: ExecuteSQL • **Method**: MakeTables

Executing a Database Procedure

To execute a database procedure, use the **sql execute procedure** statement. You can pass parameters either by name or by position. For example:

Example: executing a database procedure

```
empid : integer = 12345;
salaryIncrement : integer = 15000;

-- Passing parameters by position.
sql execute procedure updateSalary(empid, salaryIncrement);

-- Passing parameters by name.
sql execute procedure updateSalary(AddToSalary = salaryIncrement,
   Id = empid);
```

Vendor-Specific Notes on Database Procedures

The following table describes Forte's support for database procedures:

DBMS	Procedures	Named Parameters	Positional Parameters	Comments
DB2 v2	Yes	No	Yes	v2 CLI ODBC restriction
Informix	Yes	Yes	Yes	no input/output parameters
ODBC	Yes	No	Yes	drivers may provide less, eg Access
Oracle	Yes	Yes	Yes	
Rdb	Yes	No	Yes	
SQLserver	Yes	No	Yes	ODBC restriction
Sybase	Yes	Yes	Yes	

- Informix dbprocs do not support input-output parameters. If input-output parameters are passed from a TOOL execute procedure, an exception is generated.
- If the ODBC database driver you are using does not support procedures, you will receive a runtime error.

Oracle

Oracle procedures can return functional results. The following example is valid for Oracle:

```
intval : integer;
intval = IntegerData((sql execute procedure p(param1 = 5))).Value;
```

Because **sql execute procedure** returns a DataValue object, you must cast the return value to the desired class. In this example, an extra set of parameters is required in order to cast the result to IntegerData. Thus, you must know what datatype the procedure returns so as to cast it appropriately. If the cast is to the wrong class, the user receives a runtime error.

Working with ImageData Objects

When selecting ImageData from a database, the data is read using one of several Forte formats, just as is described for the ReadFromFile method for the ImageData class. Forte determines the type of image just as it does when the format SP_IF_DEFAULT is used with ReadFromFile; that is, Forte determines the type of image data by reading the start of the actual data.

When inserting ImageData into a database, the data is always written as serialized ImageData.

Using Binary Large Objects (BLOBs)

You can generally manipulate BinaryData objects (also called Binary Large Objects, or BLOBs) and large TextData objects just like other datatypes. The Forte database interface allows you to fetch, insert, and update BLOB columns (both TEXT and BYTE) using the BinaryData and TextData classes.

You need not know in advance the maximum BLOB size to select BLOB data. Forte automatically allocates enough space to retrieve or manipulate any BLOB data.

When working with binary objects, you must be sure to satisfy vendor-specific requirements. To ensure application portability, follow the recommendations below when working with binary data.

Selecting Binary Data

The following example demonstrates using a cursor to fetch BLOB from a database into a BinaryData object. It then deserializes the BinaryData object into an object of type Artist. The cursor reference BlobCurs refers to the cursor that is defined in "Defining a Cursor" on page 75.

```
GetArtist(name:string) : Artist

-- This method returns an Artist Object, given an artist
-- name.  It does this by fetching a serialized version
-- of the Artist from the ArtistBlob table in the database.

-- When it gets the BinaryData, it is assumed to contain
-- a MemoryStream object which is deserialized.

-- The method returns an Artist object, or NIL value, if no
-- artist with matching name is found.

-- This will pass any exceptions back to the invoking method
-- for display, including an exception if more than one
-- matching name was found.

-- First, get the BinaryData if it exists.
rowcount : integer = 1;
binData : BinaryData = new;
blobCurs : BlobCursor;

begin transaction

     sql open cursor blobCurs(name) on session self.MGRSession;
     rowcount =  (sql fetch cursor blobCurs into :binData);
     sql close cursor blobCurs;
     if rowcount = 0 then
          return NIL;
     end if;

end transaction;
```

```
-- Now that the data is read from the DB, deserialize
-- the MemoryStream that is in the BinaryData.
strm : MemoryStream = new;
strm.Open(accessMode = SP_AM_READ, isBinary = TRUE);
-- The UseData method will simply make the stream point
-- to the memory that has already been set up in the
-- BinaryData object.
strm.UseData(data = (pointer to char)(binData.Value),
     length = binData.ActualSize);
-- Deserialize the Artist.  Remember to cast.
painter : Artist =
     (Artist) (strm.ReadSerialized());
strm.Close();

return painter;
```

See WinDB example **Project**: WinDB • **Class**: ArtistMgrTable • **Method**: GetArtist

Inserting Binary Data

When inserting BLOB data into a database, Forte generates multiple SQL statements from the original **sql insert** statement, and, in turn, requires information about column names that correspond to the data inserted. Thus, to insert a BLOB, you must use an object of a class whose attribute names and data types correspond to the columns you wish to insert into a table.

The following code example shows serializing a Forte object into a BinaryData object, and then inserting the BinaryData object into a database column declared as a BLOB type.

```
MakeDatabase()

-- This method creates the blob table needed in the example.
-- A single table is created, with an appropriate type for
-- the target database (since each supports a different
-- type for blobs).  It is created with two columns,
-- one for the key (artist name) and one with the actual blob
-- data.  The sample data is loaded first into memory,
-- and then transferred to the blob.

-- Any exceptions that occur will be passed back to the
-- code that invokes this method.  This allows the user
-- interface to display any of the error code.

-- First, get some sample data, from a generic method.
artists : Array of Artist;
artists = self.GetSampleData();

-- The tables are assumed not to exist.  Any errors if
-- they do are simply sent back to the client. Note that
-- the names of the columns use the exact same case as the
-- attributes in the tables.  This is because Sybase
-- is case sensitive in names.  Also the table name
```

```
-- is case sensitive.

case self.MGRSession.DBVendorType is

 when DB_VT_SYBASE do
      sql execute immediate
      'create table ArtistBlob(Name varchar(30) not null, BlobValue
image not null)'
      on session self.MGRSession;
  when DB_VT_ORACLE do
      sql execute immediate
      'create table ArtistBlob(Name varchar(30) not null, BlobValue
long raw not null)'
     on session self.MGRSession;
  else do
     e : GenericException = new;
     e.SetWithParams(severity = SP_ER_ERROR,
          message = 'This example does not demonstrate using blobs
for this database.');
     task.ErrorMgr.AddError(e);
     raise e;
end case;

-- Grant privileges
sql execute immediate
  'grant all on ArtistBlob to public' on session self.MGRSession;

-- Create memory streams for the artist data, transfer them into
-- BinaryData objects, and then insert into the database.

-- Note also that the data is transferred into the artistBlobObjects
-- array. The row class of this array has attributes that
-- correspond to the columns in the ArtistBlob table. This is
-- a more portable way to handle insertion of blob data.

artistBlobObjects : array of ArtistBlobObject = new;
i : integer = 1;
while i <= artists.Items do
     strm : MemoryStream = new;
     strm.Open(accessMode = SP_AM_READ_WRITE, isBinary = TRUE);
     strm.WriteSerialized(object = artists[i]);
     -- Now transfer the stream to a BinaryData object.
     strm.Seek(0);
     artistBlobObjects[i] = new;
     artistBlobObjects[i].Name = artists[i].Name;
     strm.ReadBinary(target = artistBlobObjects[i].BlobValue);

     -- Insert into the database
     sql insert into ArtistBlob(Name, BlobValue)
```

```
          values (:artistBlobObjects[i])
          on session MGRSession;
     strm.Close();
     i = i + 1;
end while;

return;
```

See WinDB example **Project**: WinDB • **Class**: ArtistMgrBlob • **Method**: MakeDatabase

Vendor-Specific Notes on BLOB Handling

Informix

Forte does not support references to Informix BLOBs by file name or file handle.

Rdb

Because Rdb does not allow updates of BLOB data, you cannot update Rdb BLOB data using either TOOL or the Forte classes. However, you can insert BLOB data.

Sybase

There are some additional restrictions when inserting or updating BLOB data with Sybase. These restrictions are necessary since the Sybase interface does not allow direct modification of Sybase text or image columns through SQL. Text or image columns must be modified by selecting back the affected row and then using a separate interface.

The restrictions on using BLOB data with Sybase follow:

1. If you insert a row which contains a text or image column, then all of the following must be true:

   a. The non-text, and non-image columns must uniquely identify the row to be inserted. In the following example the variables intval and charval must contain values that together will uniquely identify the row inserted:

   ```
   SQL INSERT INTO T (intcol, charcol, textcol) values
         (:intval, :charval, :textval);
   ```

   b. You must explicitly name the columns to be inserted.

   Unlike the previous example, the following example will fail since it does not have a column list:

   ```
   SQL INSERT INTO T values (:intval, :charval, :textval);
   ```

   This restriction does not apply if you do not insert any text or image columns.

   c. The **values** clause cannot contain literals. For example, the following example will fail because the literal 5 is invalid in this context:

   ```
   SQL INSERT INTO T (intcol, charcol, textcol) values
         (5, :charval, :textval);
   ```

2. If you insert or update a row with more than one text or image column, then *all* the text and image values must be either less than 256 bytes, or 256 bytes or more. If some values are shorter and some longer then you must break the statement into two, one which inserts or updates the text and image columns shorter than 256 bytes, and another which updates the values 256 bytes or longer.
3. When performing an update which contains a text or image column, you must include a **where** clause which limits the update to a single row.
4. When performing an update which contains a text or image column you cannot update any column which appears in the **where** clause.

Using Forte Classes to Execute SQL

Whenever possible, it is most convenient to use TOOL SQL statements directly in your methods. However, for many applications, you will need to use the GenericDBMS database class interface. This interface lets you execute a SQL command whose exact text is unknown at compile time; it provides more flexibility, but may require more programming to handle various conditions.

You can generate SQL statements at runtime in your application when any of the following is unknown:

- the text of the SQL statement
- the number and/or data types of host variables (either for input or what will be returned by the SQL statement)
- the references to columns, tables, and views

Forte sample applications

Two Forte example applications, DynamicSQL and DynamicDataAccess, demonstrate the use of the database class interface. We use the DynamicDataAccess example to demonstrate various techniques in the remainder of this chapter. You may have used the DynamicSQL sample application to verify your database connection. It is mentioned in "Testing a Resource Name with the DynamicSQL Example" on page 32. You can find general instructions for both examples in Appendix A, "Database Example Applications."

DBSession Methods

The DBSession class (along with some other classes) provides the methods you use to execute SQL in your TOOL code. In particular, you will invoke the following DBSession methods on the DBSession object for your database session:

Method	Description
CloseCursor	Closes a cursor opened by the OpenCursor or ExtendCursor method.
CloseExtent	When you are working with multiple result sets, this method cancels the current result set so you can use the ExtendCursor method to get to the next result set (Microsoft SQL Server and Sybase only).
Execute	Executes a prepared SQL DML statement (other than select or execute procedure).
ExecuteImmediate	Executes any single SQL statement that does not return results.
ExtendCursor	When you are working with multiple result sets, this method moves the cursor to the next result set or to the return parameters for the procedure (Microsoft SQL Server and Sybase only).
FetchCursor	Retrieves a single row or a set of rows from the result set of a cursor.
OpenCursor	Executes a prepared select or execute procedure statement to produce a result set.
Prepare	Prepares a SQL DML statement for execution, allowing you to use placeholders.
PreparePositioned	Prepares a SQL DML statement that references another cursor.
RemoveStatement	Removes a prepared statement.
Select	Executes a prepared select statement and retrieves a set of rows.

Executing Single SQL Statements

The simplest way to execute SQL is to use the ExecuteImmediate method. The ExecuteImmediate method executes a single SQL statement—of any type—stored in a string or a TextData object. The only restrictions are that the statement must not contain any placeholders or return any results. (Your particular database vendor may impose additional restrictions.) The ExecuteImmediate method is useful when you need to execute a statement only once.

```
DefaultDBSession.ExecuteImmediate('create table Paintings(
  Name varchar(60), Artist varchar(60), YearPainted integer);
```

The ExecuteImmediate method provides the same functionality as the **sql execute immediate** TOOL statement. There is no particular advantage to using ExecuteImmediate instead of the corresponding TOOL statement.

Using Prepared Statements

A prepared SQL statement is one that has been parsed by the database compiler. Prepared statements offer several advantages:

- You can execute the prepared statement multiple times (without reparsing), which improves performance for the second and subsequent iterations.
- You can use placeholders in the SQL statement.
- You can use a cursor to work with rows returned by the executed statement.

DBDataSet class

When you work with SQL statements that require input values (for placeholders) or that return results, you will use objects of the DBDataSet class in the GenericDBMS library. You will use DBDataSet objects both to describe and to store data for input values and result values. Some of the following examples demonstrate the use of DBDataSet objects.

There is a distinction between how you execute prepared **select** statements and prepared **insert**, **update**, or **delete** statements. While the initial steps for processing both types of statements are the same, select statements are executed using the OpenCursor or Select methods, and the other statements are executed using the Execute method.

Executing Prepared Queries

When you use the database class interface to execute a SQL statement that returns results, you may open a cursor to process the results a row at a time, or you may retrieve the entire result set all at once using the Select method.

The following steps summarize how to use a cursor to process SQL select statements. Each step is explained in detail below using examples from the DynamicDataAccess sample application.

To execute a prepared select statement:

1. Build the SQL statement.
2. Prepare the statement.
3. Set up a DBDataSet object with input values for placeholders, if necessary.
4. Open the cursor.
5. Fetch the rows.
6. Transfer or process the data.
7. Close the cursor.

About the DynamicDataAccess Example

The DynamicDataAccess sample application provides a window that lets the end user query the database by selecting specific criteria. The application constructs the actual SQL queries and then executes them using the appropriate GenericDBMS classes.

The DynamicDataAccess example allows end users to select or insert data from a database. Figure 13 shows the Select screen for this example, as an end user might fill it in.

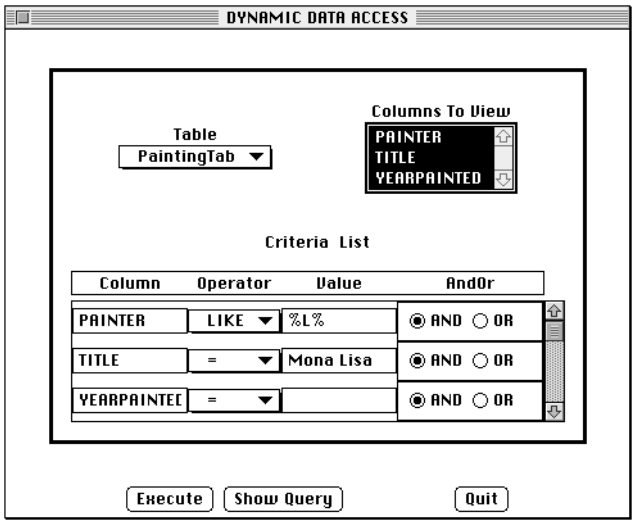


Figure 13 *Choosing Selection Criteria*

First the user chooses the table to **query**. Based on the table, the appropriate columns populate the Columns to View drop list and Criteria List array. Then the user chooses the columns to view and the criteria to use to determine the result set. When the user clicks the Execute button, Forte constructs the **sql select** statement based on the criteria just entered. In this case, Forte constructs the following SQL statement:

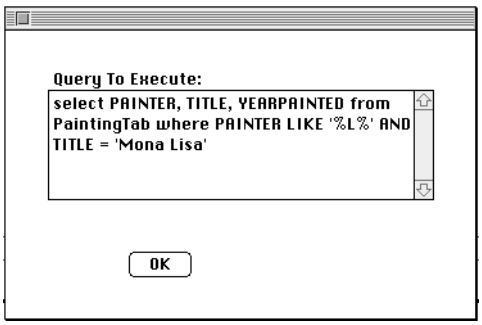


Figure 14 *Dynamically Constructed SQL Statement*

When the query executes, the application builds a screen that displays the data, as shown in Figure 15:

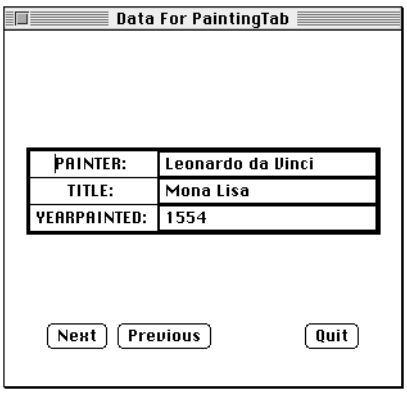


Figure 15 *Displaying the Results*

The following section discusses the SQL methods used in this application.

Building the SQL Statement

When you write an application that will generate SQL statements at runtime, you must determine the range of information that the end user may require, write a user interface that allows the user to specify the full breadth of this information, construct valid SQL statements in your methods, perform the desired processing, and possibly display or return results. To construct a SQL statement, you simply build each phrase of the query as you collect input from the user.

The following method from the DynamicDataAccess application constructs a select statement based on input from the end user, as shown in Figure 13. First, the BuildQuery method builds the select list:

Example: building the select statement

```
CommandString = 'select ';

Selected : Array of ListElement;
Selected = <ColumnList>.GetElementList(ET_SELECTED);

NumColumns = Selected.Items;
for i in 1 to Selected.Items do
```

```
  if i < Selected.Items then
   Commandstring.concat(Selected[i].TextValue).concat(', ');
  else
   Commandstring.concat(Selected[i].TextValue);
  end if;
end for;

Commandstring.concat(' from ').concat(TableList);
```

See DynamicDataAccess example

Project: QueryMgr • **Class**: QueryWindow • **Method**: BuildQuery

Then the BuildQuery method determines if there is a **where** clause:

Example: determine if there is a where clause

```
HasWhere : Boolean = FALSE;
Columns : integer = 0;
WhereClause : TextData = new;
WhereClause.concat(' where ');
for i in 1 to CriteriaArray.Items do

     if CriteriaArray[i].Value.IsNotEqual('') then
      WhereClause.concat(CriteriaArray[i].Column).concat(
          ' ').concat(CriteriaArray[i].Operator).concat(' ');
      WhereClause.concat('''').concat(CriteriaArray[i].
        Value).concat('''');
       if i < LastRow then
            WhereClause.concat('
               ').concat(CriteriaArray[i].AndOr).concat(' ');
          end if;
          HasWhere = TRUE;
       end if;
end for;
```

See DynamicDataAccess example

Project: QueryMgr • **Class**: QueryWindow • **Method**: BuildQuery

Finally, the BuildQuery method concatenates the **where** clause to the **select** statement:

```
if HasWhere then
  Commandstring.concat(WhereClause);
end if;
```

See DynamicDataAccess example

Project: QueryMgr • **Class**: QueryWindow • **Method**: BuildQuery

Preparing the Statement

Before you can invoke the OpenCursor method, you must use the Prepare method to prepare the SQL statement. The Prepare method parses the statement and gives it a unique identifier so you can later reference it in the OpenCursor method. The Prepare method also returns an integer that indicates the type of statement being prepared. If you are using placeholders, the Prepare method returns information about the placeholders used in the statement. The following example from the DynamicDataAccess application illustrates preparing a SQL statement:

Example: preparing a SQL query

```
begin
dynStatement        : DBStatementHandle;-- Statement from Prepare
inputDescriptor     : DBDataSet;        -- Description of input
outputDescriptor    : DBDataSet;        -- Description of output
outputData          : DBDataSet;        -- Actual output
statementType       : integer;
rowType             : integer;
DataFetched         : StoredData = new;

 dynStatement = MySession.Prepare(
   commandString = CommandString,    -- the command
   inputDataSet = inputDescriptor,   -- output of substitutions
   cmdType = statementType);         -- output of statement type
```

See DynamicDataAccess example

Project: QueryMgr • **Class**: ExecuteSQL • **Method**: RetrieveData

This code returns an object of type DBStatementHandle, which is used to uniquely identify the prepared statement in subsequent method invocations.

Opening the Cursor

To execute the prepared **select** or **execute procedure** statement, use the OpenCursor method. This method opens the cursor and positions it before the first row in the result set. The **inputDataSet** parameter is required in the OpenCursor method, but it is not actually used in this example because there are no placeholders. A description of the expected output is given in the **resultDataSet** parameter. Notice that this method does not actually bring any data back in the **resultDataSet**, but only describes the data that will be returned with the FetchCursor method.

Example: opening a cursor

```
rowType = MySession.OpenCursor(
  statementHandle = dynStatement,     -- the statement
  inputDataSet = inputDescriptor;
  resultDataSet = outputDescriptor);  -- the output data
```

See DynamicDataAccess example

Project: QueryMgr • **Class**: ExecuteSQL • **Method**: RetrieveData

Before you retrieve the rows, the application must determine what columns will be retrieved so it can display them properly, as shown in Figure 15. The example below uses the GetColumn method on the OutputDescriptor (which contains a description of the data) to retrieve the column description into ColumnInfo. It also stores a copy of the name in ColumnsArray, which creates the fields for the dynamically created window.

DBColumnDesc

ColumnInfo is a DBColumnDesc object. The DBColumnDesc class describes a column in a relational table (or a single placeholder, as shown in later in this section).

Example: describing the data

```
for j in 1 to NumColumns do
 ColumnName : TextData;
 ColumnInfo : DBColumnDesc;
 ColumnInfo = OutputDescriptor.GetColumn(position = j);
 ColumnName = ColumnInfo.Name.Clone(TRUE);
 DataFetched.ColumnsArray.AppendRow(ColumnName);
end for;
```

See DynamicDataAccess example

Project: QueryMgr • **Class**: ExecuteSQL • **Method**: RetrieveData

Fetching Rows from the Result Set

To retrieve data from the result set, invoke the FetchCursor method. (An example appears in the next section.) FetchCursor retrieves one row or an array of rows from the result set and stores the values in the **resultDataSet** output parameter (a DBDataSet object). You can then work with these values in your code using the GetValue method of the DBDataSet class. Each time you invoke the FetchCursor method, the cursor moves forward either one row or the number of rows you specify.

Storing the Data

The FetchCursor method uses a DBDataSet object to store the fetched row or rows. Because Forte uses the same DBDataSet object each time the FetchCursor method is invoked, if you need to preserve the data you should assign the DBDataSet values to other objects before the next FetchCursor, as in this example. For each row in the result set, the GetValue method retrieves the column values and appends them to the DataArray array. Remember, the column names have already been stored—the column values are matched by position.

Example: fetching rows from a cursor

```
numrows : integer;
    totalRows : integer = 0;
    while TRUE do
       numrows = MySession.FetchCursor(
          statementHandle = dynStatement,
          resultDataSet = outputData);
       if numrows <= 0 then
          exit;
       end if;
       TotalRows = TotalRows + 1;
       DataFetched.ResultSetArray[TotalRows] = new;
       for j in 1 to NumColumns do
          ColumnValue: TextData = new;
          (outputData.GetValue(position = j, value=columnValue));
          DataFetched.ResultSetArray[TotalRows].DataArray.
          AppendRow(ColumnValue);
       end for;
    end while;
```

storing the fetched rows

See DynamicDataAccess example

Project: QueryMgr • **Class**: ExecuteSQL • **Method**: RetrieveData

Closing the Cursor

After you have finished using the cursor, use the CloseCursor method to close it.

```
mySession.CloseCursor(statementHandle = dynStatement);
```

Executing Prepared DML Statements

You can use the DBSession class interface to execute **insert**, **update**, and **delete** SQL statements. This is particularly useful when you know what kind of statement you want to execute repeatedly, but you need to vary the criteria. Forte methods use placeholders, which are mechanisms through which you can pass data into your methods at runtime—placeholders are discussed in detail later in this section.

The following steps summarize the use of the DBSession class methods to perform **insert**, **update**, and **delete** statements. Each step is described in detail below.

To execute a prepared insert, update, or delete statement:

1. Build the SQL statement.
2. Prepare the statement.
3. Set the input values for placeholders in a DBDataSet object, if necessary.
4. Execute the statement.
5. Remove the statement if it will not execute again.

About the DynamicDataAccess Example

The DynamicDataAccess application includes a form through which users can insert data into the database, as shown in Figure 16.

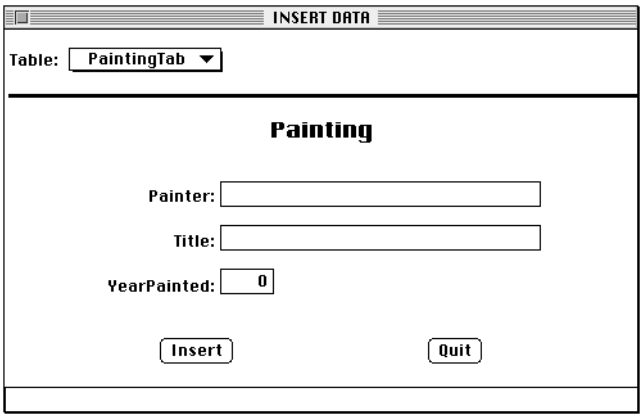


Figure 16 *Insert Screen*

Because this screen allows inserts only, the **sql insert** statement is already built in the example. This application uses placeholders to allow users to construct unique insert statements. These insert statements can also be built at runtime—see "Building the SQL Statement" on page 88. Similarly, placeholders can be used in SQL statements built at runtime.

The application first checks to see if the user selected PaintingTab or ArtistTab. Based on the user's choice, the appropriate fields are displayed. The user then enters the data to insert; this data is handled by the placeholders, described below.

Building the SQL Statement

In this particular example, we know we are building a **sql insert** statement. In the following SQL insert statement, notice that colons precede the placeholder names.

```
CommandString.setValue
  (’sql insert into PaintingTab (Painter, Title, YearPainted)
       Values (:Painter, :Title, :YearPainted)’)
```

See DynamicDataAccess example

Project:DynamicDataAccess • **Class**: InsertDataWindow • **Method**: BuildInsert

Once the statement is built, it is ready to be prepared.

Preparing the Statement

Before you can invoke the Execute method, you must use the Prepare method to prepare the SQL statement. The Prepare method parses the statement and gives it a unique identifier so you can later reference it in the Execute method. The Prepare method also returns an integer that indicates the type of statement being prepared. If you are using placeholders, the Prepare method returns information about the placeholders used in the statement—this example does use placeholders, which are discussed below.

The following code demonstrates preparing a statement:

Example: preparing a SQL statement

```
inputDescriptor : DBDataSet;
inputDescriptor = new;
  dynStatement : DBStatementHandle;
  statementType : Integer;
  dynStatement = MySession.Prepare(
        commandString = commandString,
        inputDataSet = inputDescriptor,
        cmdType = statementType);
```

See DynamicDataAccess example

Project: QueryMgr • **Class**: ExecuteSQL • **Method**: RetrieveData

The commandString parameter contains the statement that was used in the previous step. The inputDataSet output parameter contains information about the placeholders (described below), and the cmdType parameter returns the type of statement.

Value	Description
DB_CV_INSERT	Insert statement
DB_CV_UPDATE	Update statement
DB_CV_DELETE	Delete statement
DB_CV_SELECT	Select statement
DB_CV_EXECUTE	Execute procedure statement

Finally, the Prepare method returns an object of type DBStatementHandle, which is used to uniquely identify the prepared statement in subsequent method invocations.

Processing Placeholders

A *placeholder* is an arbitrary name used in a SQL statement to represent a value that will be supplied when the statement is actually executed. The placeholder values are supplied through the inputDataSet parameter (described below) to the Execute or OpenCursor

methods. For example, in the DynamicDataAccess example, the user can supply the data to insert multiple rows without the programmer having to re-prepare each **sql insert** statement.

Note

In ANSI SQL, placeholders are represented by the "?" character. In Forte, placeholders are represented by a colon preceding the variable name, as in ":name". A placeholder name can be any legal Forte name. However, if the name is the same as a SQL key word, you must enclose the placeholder name in quotation marks.

DBDataSet class

You use the DBDataSet class to represent placeholders. The DBDataSet class describes the columns and the data in the placeholders and their values.

SetValue method sets values in input row

When you use the Prepare method to prepare the statement, the inputDataSet output parameter contains information about the placeholders that were included in the DML statement. This information is provided in the form of a DBDataSet object. You can then use the SetValue method to set the values of the individual placeholders for use in a subsequent Execute method. You can use the SetValue method to set values by name or by position; setting the values by position is more efficient.

The following example demonstrates setting the values for the placeholders referenced by the inputDataSet parameter.

Example: processing placeholders

```
inputDescriptor.SetValue(columnName = ':Painter',
        value = paintingData.Painter);
inputDescriptor.SetValue(columnName = ':Title',
       value = paintingData.Title);
inputDescriptor.SetValue(columnName = ':YearPainted',
       value = paintingData.YearPainted);
```

See DynamicDataAccess example

Project: QueryMgr • **Class**: ExecuteSQL • **Method**: ExecuteInsert

The columnName parameter to the SetValue method is set to the placeholder name; the paintingData object contains the data entered by the user and used to fill in the placeholders.

Executing the Statement

After you set the values for the placeholders, you can execute the statement. Use the inputDataSet parameter of the Execute method to reference the DBDataSet object that contains the values for the placeholders. In the following example from the DynamicDataAccess application, the inputDataSet parameter references the DBDataSet object whose values were set in the previous example:

```
MySession.Execute(statementHandle = dynStatement,
  inputDataSet = inputDescriptor);
```

Note that the Execute method returns the number of rows affected, if known.

Removing the Statement

Use the RemoveStatement method when you no longer intend to execute a statement and you are finished with its result set. This is especially important if your application is complex and contains a number of statements, since each prepared statement consumes Forte resources and usually database resources as well. It may also be necessary to use exception handlers to remove statements when exceptions occur.

```
MySession.RemoveStatement(statementHandle = dynStatement);
```

To see the RemoveStatement method in context, see the Isql.Process command in the example program DynamicSQL in Appendix A.

Improving Application Performance

When you build a Forte application that accesses one or more databases, you have a number of variables to consider when tuning performance. Applications can rapidly become very complex, as they scale up to accommodate new users, new components, and more data. While tuning a Forte database application is an appropriate topic for a large book, this section contains some high level reminders that, in general, should always be heeded to maximize performance.

To maximize your application's performance, remember to:

1. Use explicit Forte transactions.

   For best performance you should use explicit Forte transactions for all SQL statements, including selects.

2. Remove a cursor after closing it.

   Unremoved cursors slowly use up the open cursors allowed by a database, and will eventually lock up the instance.

3. Wrap exception handlers around all of the prepare/fetch/close code and perform the remove in the exception handler as well. Otherwise exceptions occuring after the prepare will leave cursors unremoved.

Multithreaded Database Access

Forte supports non-blocking database connections on platforms where Forte uses native threads and the vendor's libraries are thread safe. At this time, these conditions are true only on NT with a server partition connecting to the Microsoft SQL Server or Oracle database.

Therefore, in either of these cases, a Forte server partition can have other tasks running while one task is waiting on results from the database. Thus, accessing the database from one task does not block other tasks in the partition from running. This capability may allow sites to reduce the number of replicates needed when load balancing, or eliminate the need for process-level load balancing.

Note, however, that this does not change the transactional nature of DBSession objects. A DBSession object can only participate in one transaction at a time. Thus while the partition may not be blocked and may be able to run other tasks, those tasks may not be runnable if they reference a locked resource—for example, a DBSession object being used by another task in another transaction.

Mapping DBDataSets into Objects

When you select data, it is initially returned in a DBDataSet object. While it is most efficient to use the returned object directly, it is often not very convenient. Usually you will put the selected data into one or more custom objects. You typically use the GetValue method of the DBDataSet class to move data from the DBDataSet to your object.

Comparing GetValue methods

While the GetValue method can handle any data type, it has four general variations. Performance for these variations varies. The variations are described below:

- GetValue can take either an integer *position* or *name* of the column for which to retrieve data.

  Using positions is faster since Forte can index the column directly rather than looking up a name in a list.

- GetValue can return a DBDataSet *return value* or pass an *ouput parameter*.

  Using passed parameters is faster when you have an object into which you want to retrieve the result. Using a return value requires that a new object be created to hold the value; then the new object is returned. If you will then store the value in yet another object (and discard the returned object), it is more efficient to use the variation of GetValue that returns the value in a parameter.

For example this is faster:

```
myObj : myObjType;
//  All examples assume that myObj gets initialized somehow
dataset.GetValue(position = 1, value = myObj.myAttr);
```

than this:

```
myObj : myObjType;
myObj.myAttr.SetValue(dataset.GetValue(position = 1));
```

which is faster than:

```
myObj : myObjType;
myObj.myAttr.SetValue(dataset.GetValue(columnName = 'mycol'));
```

Of course this assumes that myAttr has already been instantiated. If it hasn't the following may be more convenient:

```
myObj : myObjType;
myObj.myAttr = dataset.GetValue(position = 1);
```

Note: the preceding is a little different than the seemingly equivalent:

```
myObj : myObjType;
myObj.myAttr = new;
dataset.GetValue(position = 1, value = myObj.myAttr);
```

The difference lies in the fact that the datatype of myAttr and the column in the dataset may differ. In the latter example the GetValue method will convert the value in the dataset to the type of myAttr (if possible). In the former example you will get a type clash error at runtime if the type of myAttr isnot the same class or a superclass of the type of the column.

Vendor-Specific Information

Informix

Scroll Cursor Support

A scroll cursor must be used in an explicit Forte transaction; it cannot be used in an implicit transaction. To use a scroll cursor, use the optional parameters for the DBSession methods Prepare and FetchCursor.

An example of using Informix scroll cursor feature follows:

```
SelectStatement  : DBStatementHandle = NIL;
InputDescriptor  : DBDataSet;
OutputDescriptor : DBDataSet;
CmdType          : Integer;
MaxRows          : 3;
Rows             : Integer;

-- Prepare the statement by indicating that we want
-- to open a scroll cursor.
SelectStatement = DefaultDBSession.Prepare(
    commandstring = 'select * from test',
    inputDataSet = InputDescriptor,
    cmdtype = CmdType,
    forScroll = TRUE);

begin transaction
--Scroll cursor can only be used within an explicit transaction.
DefaultDBSession.OpenCursor(
    statementHandle = SelectStatement,
    inputDataSet = InputDescriptor,
    resultDataSet = OutputDescriptor);

-- skip 2 records, fetch the next
Rows = DefaultDBSession.FetchCursor(
    statementHandle = SelectStatement,
    resultDataSet = OutputDescriptor,
    maxrows = MaxRows,
    key = DB_FK_RELATIVE, offset=2);
printResult(OutputDescriptor);
-- The method printResult is not showed here in the example.

-- fetch the next row
Rows = DefaultDBSession.FetchCursor(
    statementHandle = SelectStatement,
    resultDataSet = OutputDescriptor,
    maxrows = MaxRows,
    key = DB_FK_CURRENT);
printResult(OutputDescriptor);
```

```
-- fetch the 8th row
Rows = DefaultDBSession.FetchCursor(
    statementHandle = SelectStatement,
    resultDataSet = OutputDescriptor, maxrows = MaxRows,
    key = DB_FK_ABSOLUTE, offset=8);
printResult(OutputDescriptor);

-- fetch the previous (7th) row
Rows = DefaultDBSession.FetchCursor(
    statementHandle = SelectStatement,
    resultDataSet = OutputDescriptor, maxrows = MaxRows,
    key = DB_FK_PREVIOUS);
printResult(OutputDescriptor);

-- fetch the first row
Rows = DefaultDBSession.FetchCursor(
    statementHandle = SelectStatement,
    resultDataSet = OutputDescriptor,
    maxrows = MaxRows,
    key = DB_FK_FIRST);
printResult(OutputDescriptor);

-- fetch the next (2nd) row
Rows = DefaultDBSession.FetchCursor(
    statementHandle = SelectStatement,
    resultDataSet = OutputDescriptor,
    maxrows = MaxRows,
    key = DB_FK_NEXT);
printResult(OutputDescriptor);

-- fetch the last row
Rows = DefaultDBSession.FetchCursor(
    statementHandle = SelectStatement,
    resultDataSet = OutputDescriptor,
    maxrows = MaxRows,
     key = DB_FK_LAST);
printResult(OutputDescriptor);

DefaultDBSession.CloseCursor(statementHandle = SelectStatement);
end transaction;
```

Chapter 6

Transactions

When you include SQL statements (or equivalent methods) in a Forte application, you use the Forte transaction model to start explicit transactions or under some circumstances, implicit transactions.

This chapter describes how to use transactions so that your application uses transactions properly and with the best possible performance. A rule of thumb is to always use explicit Forte transactions for best performance.

This chapter discusses a variety of transaction scenarios, and includes the following topics:

- explicit and implicit transactions
- dependent and independent transactions
- common problems with shared and transactional objects
- two-phase commit and distributed transactions
- vendor-specific notes

Relationship of Forte and Database Transactions

When you write a Forte application that accesses a database, you will structure your application into Forte *transactions*. (Forte transactions are the same as TOOL transactions.) Like a database transaction, a Forte transaction should consist of a set of statements that should be treated as a unit, succeeding or failing together.

Use Forte transactions, not database transactions

The only transaction commands you use are to start and end Forte transactions. You do *not* use the explicit transaction statements for your database (such as commit, rollback, or set autocommit), because Forte starts and coordinates all database transactions on your behalf. You can, however, use the Forte Abort method to abort both the Forte and DBMS transactions. Thus, you need only be concerned with Forte transactions, because all database transactions are transparently managed.

Always use explicit transactions

You should always use explicit transactions. If you do not, Forte will start implicit transactions and your application's performance may be significantly impacted. The following sections describe explicit and implicit transactions in more detail.

Local transactions perform best

A transaction that is entirely local provides better performance. A transaction that starts and ends in the same partition and makes no intervening remote calls is a local transaction. A transaction becomes distributed when a remote call is made. To the degree that you can increase the number of local transactions in an application relative to distributed transactions, you can improve the performance of the application.

Explicit Forte Transactions

You should always include SQL statements and methods in explicit Forte transactions. Your application will perform far better if you do so, particularly when you select data using a cursor (this includes TOOL **sql select** statements where the target is an array object). In addition, some operations behave differently when not used in an explicit transaction (see the Forte online Help).

Starting a Forte transaction

To start an explicit Forte transaction, you can do either of the following:

- use the **begin transaction** TOOL statement (described in the *TOOL Reference Manual*)
- use the TransactionHandle class (described in the Forte online Help)

Ending a Forte transaction

To end a Forte transaction, you use the **end transaction** clause of the **begin transaction** statement, or you can use methods on the TransactionHandle class. A Forte transaction is always ended explicitly; no type of event or statement will implicitly end a Forte transaction.

Any number of SQL statements can be contained in a transaction; however, the transaction should consist only of statements that should succeed or fail together — no other statements. You should also exclude any SQL statements that your database does not allow within a transaction (for example, some systems do not allow DDL statements in transactions).

The following example shows two SQL statements enclosed in an explicit transaction defined using TOOL:

Example: begin transaction statement

```
begin transaction
  sql insert into mytab (a) values (10) on session MySession;
  sql insert into othertab (b) values (12) on session YourSession;
  exception
    when e : GenericException do
      task.ErrorMgr.showErrors(TRUE);
end transaction;
```

When you embed SQL in an explicit Forte transaction, Forte automatically starts a database transaction, and both the Forte and database transaction must succeed together or both will fail. That is, if failure occurs in the underlying database transaction, it rolls back and the enclosing Forte transaction also aborts. Or, if the enclosing Forte transaction aborts, Forte rolls back the underlying database transaction.

Implicit Forte Transactions

If you invoke a method on the DBSession object or issue a TOOL SQL statement that accesses the database, and no Forte transaction is currently in effect, then Forte starts an implicit Forte transaction. While an implicit transaction does ensure the integrity of your data with respect to the database, it can also have a severe impact on performance. An implicit transaction is slower and may consume more memory than an explicit transaction.

An implicit Forte transaction always has a scope of one SQL statement. Execution of multiple SQL statements enclosed in separate transactions is generally slower than if the statements are executed within a single transaction. While this may be acceptable for testing or ad hoc use, implicit transactions should never be used in deployed applications because they dramatically reduce performance.

Single SQL DML Statements

If you issue a single SQL statement, such as **sql delete**, **sql insert**, **sql update**, and **sql execute immediate**, Forte places the statement in an implicit Forte transaction encompassing just the SQL statement. That is, the transaction will be started before the SQL statement and committed after it. No other TOOL statements are included in the transaction. For example:

```
-- An insert
sql insert in paintings values (:newPainting);
```

SQL Execute Immediate Statements with DDL

The **sql execute immediate** statement can be used to issue data definition language (DDL) statements, such as create table. The database vendors treat DDL statements differently with respect to transactions, as shown in the following table:

Database	Interaction between a DDL statement and a transaction
DB2	Allows one or more DDL statements in a multi-statement transaction.
Informix	Allows one or more DDL statements in a multi-statement transaction.
ODBC	Uses the semantics of the underlying database.
Oracle	Use of a DDL statement automatically commits the current transaction.
Rdb	Allows one or more DDL statements in a multi-statement transaction.
Sybase	Allows only one DDL statement in a transaction.

If you need to ensure portability across databases, do not put **sql execute immediate** statements containing DDL in Forte transactions that contain more than one SQL statement. Instead, enclose each DDL statement in an independent transaction block.

Use of Cursors in Implicit Forte Transactions

While you can use cursor statements in implicit transactions, it is not recommended. Forte buffers the entire result set for a cursor, which consumes memory. As a result, performance may degrade because Forte retrieves the entire result set before executing the next statement. The following sections describe the effect of implicit Forte transactions on the TOOL SQL statements **sql open cursor**, **sql fetch cursor**, and **sql close cursor** . In general, the descriptions below also apply when you use the OpenCursor, FetchCursor, and CloseCursor methods on a DBSession object.

SQL Open Cursor

A **sql open cursor** statement executed outside an explicit TOOL transaction is enclosed in an implicit transaction. If a cursor is opened in an implicit transaction, the **sql open cursor** statement alone causes all the data to be fetched, buffered, the database cursor to be closed, and the implicit transaction to end. The database cursor is closed because most databases do not allow a database cursor to remain open after the transaction ends. However, the TOOL cursor remains open and subsequent **sql fetch cursor** statements retrieve results from the cursor as usual, but the data is retrieved from the buffer, rather than directly from the database.

Executing **sql open cursor** within an implicit transaction has several effects, all of which can degrade performance:

- The entire result set is retrieved for the cursor, and data is copied twice: once into the buffer, and again into wherever you finally put it.
- It requires additional memory, as the entire result set must be buffered in memory at one time.
- Control is not returned back to the user to start processing the first row of data until all of the data is retrieved from the database.

Underlying data may be updated with no notification

In addition, because the implicit transaction automatically buffers all the data and commits the transaction, the actual data in the database corresponding to the buffered data may be concurrently updated without any notification to your TOOL program, or any locking on the data. This is not a problem for read-only data, but for updatable data, you should be sure to consider a concurrency strategy that is appropriate. For maximum consistency, use explicit Forte transactions.

SQL Fetch Cursor

The **sql fetch cursor** statement is never enclosed in an implicit transaction. The behavior of a **sql fetch cursor** statement executed outside of an explicit Forte transaction depends on when (how) the cursor was opened:

- If the cursor was opened in an explicit Forte transaction, then the cursor was closed at the end of the transaction; any fetch outside the transaction in which the cursor was opened will fail.
- If the cursor was opened outside of an explicit transaction, the result set for the cursor was buffered and the fetch statement will fetch the next row from the buffer.

The following example shows two cursors: cursor1 is opened in an implicit transaction and cursor2 in an explicit transaction:

Example: sql fetch cursor

```
sql open cursor1...-- surrounded by an implicit transaction

begin transaction
sql open cursor2...
while condition do
  sql fetch cursor2...-- fetches a row from the database
  sql fetch cursor1...-- fetches a row from the buffer
  other TOOL statements;
end while;
end transaction;

sql fetch cursor2...
-- This will cause an exception because the
-- cursor was closed at the end of the transaction.

sql fetch cursor1...
-- This will execute successfully because it continues
-- to fetch from the buffer.
sql close cursor1...
-- This will execute successfully and change the state of
-- the cursor to closed so it may subsequently be opened again.
```

Note that when using the FetchCursor method on the DBSession object on a cursor that was opened in an implicit transaction, you should not use the maxrows input parameter to determine the number of rows returned. Instead, check the functional result. Generally rows are returned one at a time.

For Loops

When you use a **for** loop with a cursor name or **sql select** statement, Forte performs an open cursor as part of the initialization of the **for** loop. If no explicit Forte transaction is in effect, this open cursor is enclosed in an implicit transaction, causing the result set to be buffered as described earlier in the "SQL Open Cursor" section. However, the body of the **for** loop is not part of the transaction.

SQL Close Cursor

When you issue the **sql close cursor** statement against a buffered cursor, the data buffered for the cursor is released.

Independent, Dependent, and Nested Transactions

It is strongly recommended that you always use transaction blocks for units of work that might be a transaction.When you start an explicit transaction, you can make it one of three types of transactions:

- independent
- dependent
- nested

For transactions that interact with databases, you will most often use dependent, and in some cases independent. (Dependent transactions are the default if you do not specify a type in the **begin transaction** statement.) You should not declare nested transactions that manipulate database data (see below for an explanation).

Using Dependent Transactions

A coding practice that makes transaction design more modular is to define as a dependent transaction any segment of code which might comprise a unique transaction in some context. This has the dual benefits of ensuring that all SQL always runs in an explicit transaction with no constraint on its re-use. In fact, the default type of transaction is dependent; the statement **begin transaction** is the same as **begin dependent transaction**.

An example will help illustrate. Assume a method containing the following code fragment:

```
begin dependent transaction;
SQL UPDATE personTable set dept = :var where dept = :cond;
```

This code can run as a top level transaction or it can be invoked from a context where a transaction is already in progress:

- If it is invoked when a transaction is not in progress, it simply starts a new transaction.
- If it is invoked when another transaction is in progress, it would just become part of the transaction already in progress.

However, if instead you used **begin independent transaction**, this method could not be invoked when a transaction was already in progress, since an independent transaction can not be nested inside another transaction.

Using Independent Transactions

Since dependent transactions can be used whether or not a transaction is in progress, when are independent transactions preferred? While dependent transactions are appropriate for many situations, there are circumstances when you should start a new transaction, rather than pick up or continue from a previous step. In such a case you should explicitly use an independent transaction (using the TOOL statement **begin independent transaction**, for example).

Avoid Nested Transactions

While the preferred way to execute all SQL statements (using TOOL or equivalent methods) in a Forte application is within an explicit Forte transaction, you should avoid placing SQL statements within *nested* Forte transactions. While Forte does not flag nested Forte transactions containing SQL as an error, using this construction can result in inconsistent data for the following reasons.

While Forte nested transactions provide a useful mechanism for fine-tuning the scope of a rollback in applications with complex transactional structure, most database vendors do not support nested transactions. The result of this fact is that SQL statements in a nested Forte transaction are considered *part of the outer Forte transaction*, not the nested transaction. (All other transactional operations within the nested transaction succeed or fail together as the nested transaction.) This has the following implications:

- If the nested Forte transaction aborts, but the outer Forte transaction commits, the SQL work is committed.
- Only if the *outer* transaction aborts is the SQL work aborted or rolled back.
- If the nested transaction aborts, the SQL statements within the nested transaction are not rolled back, as long as the outer transaction commits. However, other Forte transactional logging which may have occurred within the nested transaction *is* rolled back, potentially leaving an inconsistent data state.

This behavior is true regardless of whether the nested transaction is invoked locally or remotely relative to the outer transaction, and whether the transaction is within a bracketed transaction block or asynchronous nested task.

Common Problems with Shared and Transactional Objects

Because the DBSession object is both a shared and transactional object, you must take the same precautions that you would for other shared and transactional TOOL objects. The most common problems are described below—note that all these problems can occur in normal TOOL objects with the same properties.

In general, you should avoid starting transactions before any operation that might wait—for example, before an event loop where your application might wait for user input. If you begin a transaction in such a situation, you might unnecessarily cause data to be locked, preventing access by other users.

Unexpected Blocking Due to a Long-Running Query

While executing a query in a database session, the DBSession object holds the internal mutex lock on the session. While this mutex is held, no other query may be executed. Note that due to the nature of Forte's database interface, the entire Forte server process may be blocked while the query is running.

Unexpected Blocking Due to a Long-Running Transaction

If a task begins a transaction and then executes a TOOL SQL statement against a DBsession object, that session's transaction lock is now exclusively held by the task's transaction. The lock remains held until the transaction terminates (commits or aborts). No other tasks may execute any SQL statements against that same session, unless those tasks are in the same transaction or the owning transaction terminates. For example, assume the following code fragment uses a DefaultDBSession:

```
begin transaction
  event loop
    SQL select...into array object;
    when <detail>.click do
      start task self.SelectCurrentRow(currentRow)
        where transaction = dependent;
    when <update>.click do
      SQL...
    when <save>.click do
    exit;
    when <abort>.click do
      transaction.Abort(TRUE)
  end event;
end transaction;
```

Note that an event loop begins after the transaction has begun. Because an event loop can potentially last a long time, any data that is locked by a given SQL statement will remain locked until you exit the window.

Any child transaction is dependent on the parent task's transaction; thus, any queries executed within the child task are not blocked. However, queries executed by other tasks not participating in the same transaction (or from other clients) are blocked.

Avoiding Deadlocks

It is possible under some scenarios to encounter a database deadlock. These scenarios most often entail multiple database sessions accessing the same table data on the same server. For example:

- If an application has two or more database sessions accessing the same table on the same server (but different rows), a deadlock may occur on a database that uses page-level locking (Sybase Adaptive Server 11.5, for example).

Note Starting with Adaptive Server 11.9, Sybase provides row-level locking.

- If an application has two or more database sessions that are not only accessing the same table on the same server, but also the same rows, then a deadlock may occur on a database that uses row-level locking (Oracle, for example).

The fundamental basis for the deadlock is that the database cannot detect that separate Forte sessions are part of the same transaction. These scenarios may result in deadlocks, whether the database sessions originate from one or more clients, from one or more Forte transactions, or from one or more partitions (or from multiple applications).

You can usually write an application so as to avoid deadlocks. Depending on the requirements of the application, the following hints may be useful:

- Use a single database session to access data. Multiple requests within the same transaction and same database session do not block.
- When using multiple database sessions, try to ensure that each session accesses different tables.
- When using multiple database sessions and accessing the same data, an application should minimize contention at the database level by synchronizing the invocation of transactions. This can be accomplished by using a Forte transaction lock to block and defer other transactions at the Forte level.
- When possible, narrow the scope of each transaction to encompass the fewest SQL statements using a single database session. This minimizes the possibility of deadlock, since locks held are immediately released upon completion of the explicit transaction.

If a database deadlock occurs, the database will raise a DBDeadlockException to the Forte application. Your application should be prepared to handle this potential exception.

Transactions and Database Sessions

A single Forte transaction can access more than one database session, more than one database, or more than one database resource manager. Forte acts as the transaction coordinator for all Forte transactional objects, coordinating all commits and aborts.

For example, if a transaction accesses two resource managers and one resource manager aborts the transaction, Forte notifies the other database that the transaction is aborted. Likewise, when the transaction commits, Forte signals all resource managers to commit the transaction.

While Forte coordinates transaction commit across all of the sessions using a two-phase commit protocol, the Forte database interface cannot currently use DBMS two-phase commit mechanisms. The Forte transaction manager will commit all database transactions first, and then the recoverable object transactions. This means that you should plan for failures of multi-session commits accordingly.

Multitasking in a Database Session

If you want to use the same database session in multiple concurrent tasks, all the tasks must be in the same transaction. You can add a task to a transaction in progress in one of two ways:

- Use the **dependent** option on the **start task** statement.
- Use the Join method on the TransactionHandle class.

See the *TOOL Reference Manual* for more information on transactions and multitasking

Forte Support for Two-Phase Commit

Forte's support for two-phase commit (2PC) is limited by general database vendor implementation restrictions on multiple-vendor, two-phase commit. Forte guarantees two-phase commit with respect to Forte objects; however, Forte only guarantees two-phase commit with respect to database data if a third-party transaction monitor, like Encina or Tuxedo, is used in conjunction with Forte. (See "Two-Phase Commit with Multiple Database Vendors" on page 111.)

What is two-phase commit?

The purpose of *two-phase commit* is to allow a transaction to span multiple databases, regardless of vendor, with a guarantee that the transaction is committed only after all databases involved in the transaction guarantee that the commit will succeed. For example, the following "code" would use two-phase commit:

```
begin transaction
sql update ... (on dbms 1);
sql update ... (on dbms 2);
end transaction;
```

Forte's two-phase commit implementation

The main purpose of two-phase commit in a distributed transaction manager is to enable recovery from a failure that occurs *during the actual transaction commit processing*. The Forte transaction manager was built with this in mind, but only with respect to the "volatile" (or "in memory") objects that Forte manages. What this implies is that because Forte stores objects in memory and not persistently on disk, the requirement of recovery for these objects is significantly reduced.

In the Forte distributed two-phase commit model, tasks and messages carry along with them transaction identification and, during commit processing, every distributed participant is polled for its availability to commit the transaction. If an application saves persistent data to disk during a distributed Forte transaction, that application should address the potential for failure during the transaction commit processing.

Possible scenarios during the "commit window"

Forte's prepare phase polls each site (confirming a communications link with each distributed participant) but no prepare request goes to the database. When all sites are ready to commit, *Forte expects that the commit will complete successfully.* Then, at this point, called the "commit window," three scenarios can occur:

- All participating servers commit successfully. The distributed transaction is committed, and all data is consistent.
- One participating server terminates (with data not yet committed) but no other participating server has committed its unit of work. In this scenario, Forte will abort the transaction, and the data will not be inconsistent. The entire transaction is, in effect, rolled back.
- One participating server terminates (with data not yet committed) while a second participating server has already committed its unit of work. In this scenario, the outcome of the distributed transaction is inconsistent data.

Only the last scenario is cause for concern. Without two-phase commit, if one server fails during the process of committing (that is, while TOOL is executing the **end transaction** clause), there is no guarantee that the other server(s) will abort, as they might have already committed. Therefore, if a transaction is being committed and it accesses multiple database sessions, multiple databases, or multiple DBMS resource managers and there is a failure during the commit window, your data might be inconsistent. You should take this into consideration when planning your transactions.

Mission critical applications that require distributed transactions can anticipate the third "commit window" scenario in a number of ways:

- Utilize a TP monitor such as Encina (see below)
- Log distributed updates in an auxiliary database table (much like a distributed transaction monitor's transaction-state log). While this approach has been the traditional banking application solution prior to the commercial availability of products like Encina, it is somewhat complex. Nor is it generic enough so as not to have to change code every time a new table or database site is introduced into the data model.
- Rearrange the data model in order to eliminate the need for distributed transactions. This is usually only a temporary solution (with smaller numbers of active clients) and cannot be applied to complex legacy systems.

With the advent of the X/Open distributed transaction architecture (the XA Interface) more database vendors have found that by complying with the XA interface they can plug their database-specific implementation of transaction into a globally managed transaction, with commit and abort processing being conducted by a central coordinator. Of course, the overall transaction manager coordinating the global transaction must itself, persistently record the state of the different distributed branches participating in the transaction. A significant portion of the functionality provided by products such as Encina is to provide exactly this global transaction management.

Two-Phase Commit with One Database Vendor

You can use a true implementation of two-phase commit in a Forte application if both of the following conditions are true:

- The distributed transaction is between two or more databases from the same vendor.
- The distributed transaction is initiated by the SQL statements issued within one database, which in turn calls another database to complete the transaction.

In this scenario, the two-phase commit protocol of the RDBMS itself manages the transaction across the multiple databases; Forte could be considered "outside" of the actual distributed transaction.

Two-Phase Commit with Multiple Database Vendors

Encina is required for 2PC across database vendors

Two-phase commit with multiple databases, except for the limited case above, requires the use of Encina with Forte (see *Integrating with External Systems*). To use Encina you must use an XA-compliant TP monitor (Encina) along with the two XA-compliant database systems.

Notes on Vendor-Specific Transaction Handling

Generally, you should avoid using explicit database transaction statements, as Forte starts database transactions for you. And, as described in "SQL Execute Immediate Statements with DDL" on page 102, you should only execute SQL DDL statements individually, in explicit independent transactions. However, some transaction control statements are allowed.

Set autocommit on is not supported

Forte does not support the use of autocommit. Do not use the statement **set autocommit on** within a **sql execute immediate** statement.

DB2

DB2's prepared SQL statements do not persist across transaction boundaries. Forte will automatically re-prepare TOOL statements as required. To minimize this overhead, you should use multi-statement, explicit transactions.

Informix

The Forte interface to Informix IDS Server supports connections to each type of Informix database:

- a non-ANSI database with logging
- a non-ANSI database without logging
- an ANSI database

While Informix handles transactions and logging differently in these databases, Forte transparently accesses any database, and issues (or ignores) transactional SQL statements as appropriate for that database type.

Note

NOT WAIT is the default for Informix's SET LOCK MODE; any lock conflicts encountered will return an error immediately.

Rdb

If you use multiple database sessions within a Forte transaction, take care to avoid locking conflicts between the sessions, because Rdb does not allow for the sharing of locks within an Rdb process. This is a concern only if the sessions are running within the same Forte server partition.

By default, all Rdb transactions initiated through the Forte application interface are in "READ WRITE NOWAIT" mode.

To set an Rdb database session to a different transaction mode, you can issue a **sql execute immediate** statement in your application code to issue Rdb **DECLARE TRANSACTION** or **SET TRANSACTION** statements. Because these statements are valid only if there is no active Rdb transaction, you must issue these statements immediately after a TOOL **begin transaction** statement. For example:

```
begin transaction
    sql execute immediate 'set transaction read only';
...
end transaction;
```

Alternatively, a DECLARE TRANSACTION statement issued implicitly (outside of a TOOL transaction) will affect the mode of a subsequent Rdb transaction, either implicit and explicit. For example:

```
-- Read-only SQL operations
sql delete from emp;
```

The **sql delete** statement will fail because its implicit transaction is in READ ONLY mode.

Chapter 7

Error Handling

Forte converts all database errors into exceptions so that a single exception handler can process all errors that occur in a set of SQL statements. This allows you to separate error handling code from the body of a program, providing for more readable and maintainable code.

This chapter contains a brief overview about how Forte handles database exceptions.

See the Forte online help (under Native Database Errors) where you can find, for each supported database, a mapping of native database errors to Forte exceptions. The online help also lists the ODBC and ISO SQL-92 mappings to Forte exceptions.

The Exception class reference can be found in the Forte online help.

Types of Database Exceptions

The class hierarchy for exceptions that may be returned in a Forte database application is shown in Figure 17.

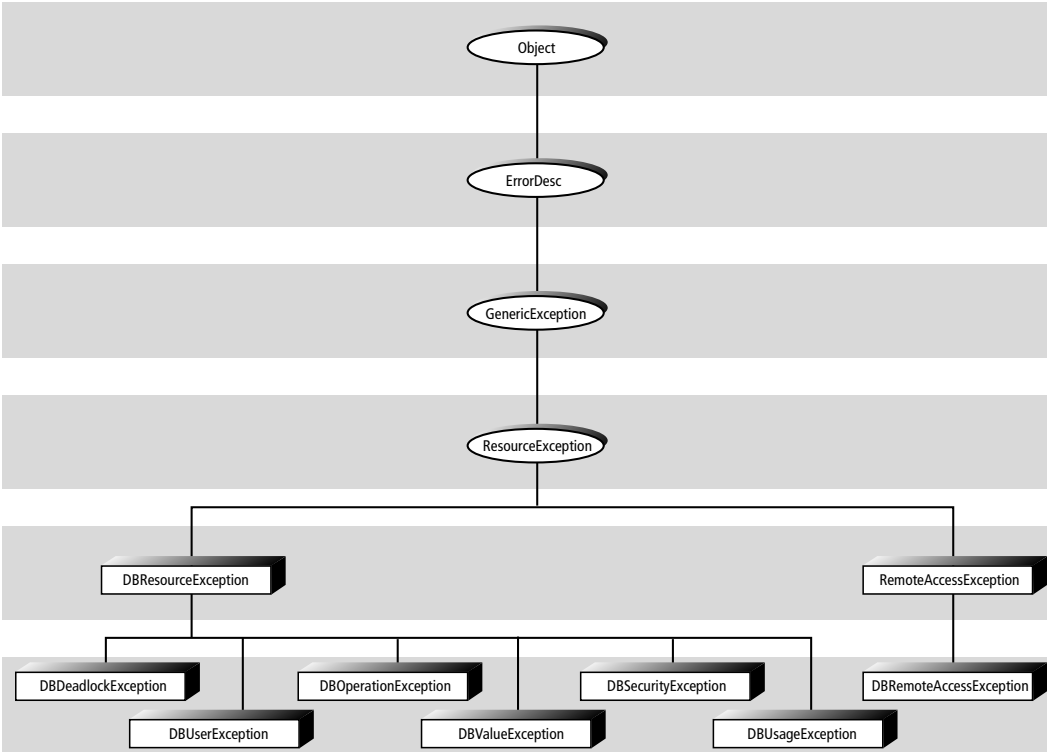


Figure 17 *The Database Exception Hierarchy*

Forte uses two general classes for database exceptions: DBRemoteAccessException and DBResourceException.

DBRemoteAccessException class

Exceptions of class DBRemoteAccessException occur when a session cannot be initiated or a current session is no longer valid. These types of errors are generally non-recoverable in the sense that you lose the current transaction and session if one existed. You can recover in the sense that you can attempt to start a new database session, or connect to a different server. Refer to the Forte online Help for more information on this exception class.

DBResourceException class

Exceptions of class DBResourceException occur when accessing a database. These types of errors are considered recoverable, since the current transaction or session is not lost, and the application can continue after handling the exception. See the Forte online Help for more information on this class.

Subclasses of DBResourceException

Many common database errors result in exceptions of type DBResourceException, such as incorrectly formed SQL statements, data constraint violations, access violations, and so on. In fact, there are several subclasses of the DBResourceException class as shown below:

Exceptions	Cause
DBDeadlockException	A database deadlock.
DBOperationException	An error in the runtime operation of a SQL statement.
DBSecurityException	A security violation.
DBUsageException	Incorrect TOOL program use for the (particular) database: syntax errors, attempt to use non-existent tables, and so on.
DBUserException	An error generated by a user database procedure or trigger.
DBValueException	An error in assigning a value, either through conversion, truncation, constraint, or arithmetic error.

See the Forte online help for more information on each of these subclasses.

DBMS errors and Forte exceptions

The Forte online help (Native Database Errors) contains tables that map vendor-specific errors to these exception classes. Refer to the table for your database vendor to become familiar with how errors generated in your database map to the various Forte exceptions.

Appendix A

Database Example Applications

Forte provides a number of example applications that illustrate how to use the features described in this manual. This appendix provides instructions on how to install the examples, a brief overview of the applications to help you locate relevant examples, and a section describing each example in detail. Typically, you run an example application, then examine it in the various Forte Workshops to see how it is implemented. You can modify the examples if you wish.

How to Install Forte Example Applications

You can access the Forte example applications only if they have been installed into your central repository or into a private local repository during installation of Forte, or if you have imported them into your repository.

The examples are located in subdirectories under the FORTE_ROOT/install/examples directory. The example applications are stored as .pex files. If they are not already installed in your repository, import them by including the tstapps.fsc script in Fscript. The tstapps.fsc script is located in the FORTE_ROOT/install/examples/install directory. Bring up Fscript in standalone mode and issue the following commands:

```
fscript> UsePortable
fscript> SetPath %{FORTE_ROOT}/install/examples/install
fscript> Include tstapps.fsc
```

This will import most of the example applications and quit Fscript. Note that certain highly specialized examples are not automatically imported by tstapps.fsc.

To run an application, select it in the Repository Workshop's plan browser and then click on the Run button.

If you want to remove all the examples from your workspace, you can do so by including the remprj.fsc script in Fscript. Bring up Fscript in standalone mode and issue the following commands:

```
fscript> UsePortable
fscript> SetPath %{FORTE_ROOT}/install/examples/install
fscript> Include remprj.fsc
```

This will exclude all the example applications and quit Fscript.

Overview of Database Example Applications

This section provides an overview of the database example applications. For a complete list of the Forte example applications, see *A Guide to the Forte 4GL Workshops*.

The margin note for the following table is the name of the subdirectory under FORTE_ROOT/install/examples where you can find the .pex files for the examples. For the complete description of an individual application, see "Application Descriptions" on page 122, which lists the applications in alphabetical order.

GenericDBMS Library Examples

database/

Example	Description
DynamicDataAccess	Shows how to build dynamic SQL queries.
DynamicSQL	Illustrates the use of the GenericDBMS classes with a command line utility.
QueryMgr	Acts as a server for DynamicDataAccess.
WinDB	Illustrates the use of GenericDBMS classes in a window-based application.

Application Descriptions

This section lists the example applications in alphabetical order. Each example has five sections describing it.

The **Description** section defines the purpose of the example, what problem it solves, and what TOOL features and Forte classes it illustrates.

The **Pex Files** section gives you the subdirectory and file names of the exported projects. The examples are in subdirectories under the FORTE_ROOT/install/examples directory. You can import example applications individually if you wish. When multiple .pex files are listed, there are supplier projects in addition to the main project. You will need to import all the files listed to run the application. Import them in the order given so that dependencies will be satisfied.

The **Mode** section indicates whether the application can be run in either standalone or distributed mode, or whether it must be run in distributed mode.

The **Special Requirements** section identifies whether you need a database connection, an external file, or any other special setup.

Finally, the **To Use** section tells you how to step through the application's functions.

See the *Forte 4GL System Management Guide* if you need directions to set up a Forte server. See Chapter 3, "Making a Database Connection" if you need information on how to make a connection to a database. The database examples run against either Sybase or Oracle.

DynamicDataAccess

Description DynamicDataAccess lets you construct ad hoc SQL queries. The data fields used to display the data and the text graphics used for labels are created at runtime to match the columns in the dynamic query.

Pex Files database/querymgr.pex, database/dda.pex.

Mode Distributed only.

Special Requirements Database connection. The files artist.dat and painting.dat must be located in FORTE_ROOT/install/examples/database.

To use DynamicDataAccess:

1. Before running DynamicDataAccess, open the Project Workshop for the QueryManager project.
2. Open the DBResourceMgr service object properties dialog. Make sure the Database Manager value is correct for your database connection.
3. Run DynamicDataAccess. You will be prompted for a User Name, Password, and Database. You must provide all three values.

   This application will create its own tables and data in the database you selected. The data is read in from the files FORTE_ROOT/install/examples/database/artist.dat and painting.dat.
4. Choose Select from the radio list to select from existing artist and painting values. Choose Insert to add your own data. Follow the prompts in the Insert and Select Windows to construct SQL statements and view the results.

DynamicSQL

Description DynamicSQL is a command line utility illustrating the use of the GenericDBMS Library classes. It lets you perform standard SQL database access commands.

Pex Files frame/utility.pex, database/dynsql.pex.

Mode Distributed only.

Special Requirements Database connection, table creation (if necessary).

▶ **To use DynamicSQL:**

1. You need an environment that has a node with a resource manager. Before running DynamicSQL, open the DynamicSQL Project Workshop. Open the AnyDBMgr service object property sheet. Make sure the Database Manager value is correct for your database connection.
2. DynamicSQL does not create any tables. You must either create them ahead of time, or use an existing test table. If you need to create a test table, the files maketst.syb and maketst.ora are provided in FORTE_ROOT/install/examples/database. If you will be using maketst.syb, edit the first line to use an existing database. For example, create a database called testapps, then edit maketst.syb to start with:

   ```
   use testapps
   ```

   Use the standard mechanism for redirecting the maketst file to load the data into your database.
3. Run DynamicSQL. You will need to know the name of the database you will use, and a userid and password. Make sure that the Forte Launch Server - Console window is visible. It is this application's only window.
4. At the prompts enter your database name, then your userid and password. You should then see a SQL> prompt. If you used the maketst file, enter:

   ```
   SQL> select * from alltypes;
   ```
5. You should see the data displayed. Then enter:

   ```
   SQL> select * from alltypes where intcol = :a;
   Enter values for the following placeholders:
   a:> 1
   Reexecute? (Y/N)y
   Enter values for the following placeholders:
   a:> 5
   Reexecute? (Y/N)y
   ```
6. To end your session, type:

   ```
   SQL> exit
   ```
7. The program will print out "Disconnecting," but you will need to return the focus to the Forte Workshop yourself.

WinDB

Description WinDB uses the GenericDBMS Library classes. It lets you perform standard SQL database access commands. It also illustrates how to send and retrieve Binary Data (BLOBs) from a database, and how to read and write serialized data to a file.

Pex Files database/windb.pex.

Mode Distributed only.

Special Requirements Database connection. The files artist.dat and painting.dat must be located in FORTE_ROOT/install/examples/database.

To use WinDB:

1. You need an environment that has a node with a resource manager.

   Before running WinDB, open the WinDB Project Workshop, and open the MySession service object. Provide the correct values for your database in the Database Manager, Database Name, User Name, and User Password fields, then click OK. WinDB creates its own tables in the database you specified. The data is provided from files in the FORTE_ROOT/install/examples/database directory, called artist.dat and painting.dat.

2. Start the application. In the radio list, Table is selected by default. Click on the Make Database button. The following painter names are valid:

   ```
   Leonardo da Vinci
   Henri Rousseau
   Edgar Degas
   Jaspar Johns
   Pablo Picasso
   ```

3. Enter a valid painter name and click the Select button. (Note that additional painting data is available for Edgar Degas and Leonardo da Vinci only.)

4. Enter a valid first letter of a painter name followed by% (such as E%), and click the Select button.

5. Enter an invalid painter name and click the Select button.

6. Click on the Drop Database button.

7. Now select Blob from the radio list, click on the Make Database button, and make the same selections as you did for Table. Click the Drop Database button when you're done.

8. Now select File from the radio list, and click on the Make Database button. This will actually create a file to which objects are written. This time, when you select artists, you must type their entire name. Again, click the Drop Database button when you're done.

Appendix B

TOOL SQL Statement Reference

This appendix is the primary reference for the TOOL SQL statements:

- sql close cursor
- sql delete
- sql execute immediate
- sql execute procedure
- sql fetch cursor
- sql insert
- sql open cursor
- sql select
- sql update

These statements are a subset of all TOOL statements. The *TOOL Reference Manual* provides complete reference regarding TOOL language components and usage, and all other TOOL statements.

In general, any operation you can perform using TOOL SQL you can also perform using a GenericDBMS class. However, for all databases except Sybase, you must use the TOOL SQL statement execute procedure to execute a database procedure; you cannot execute procedures using classes in GenericDBMS.

Note on Vendor-Specific SQL Extensions

You can include vendor-specific extensions to the ANSI SQL syntax in your TOOL SQL statements. However, if you include vendor-specific clauses in TOOL statements, your code will be valid only for database systems that support that particular syntax extension.

You can make vendor-specific code conditional upon the type of database for the current database session; see “Using Conditional TOOL for Vendor-Specific Code” on page 70 for information on using vendor-specific extensions in the Forte SQL statements.

A small number of vendor-specific database features are explicitly not supported; see “Unsupported Database Features” on page 22.

Because the Forte parser does not explicitly recognize vendor extensions, it cannot verify whether they are syntactically or semantically correct. Also, TOOL code that compiles may run without exception against one database but not against another, due to different support for the SQL language.

SQL Close Cursor

The **sql close cursor** statement closes a cursor that was opened with the **sql open cursor** statement.

Syntax

sql close cursor *cursor_reference*;

Example

```
sql close cursor dbcursor;
```

Description

After a cursor is closed, you cannot use it again until you open it. If you close a cursor before all rows have been fetched, the remaining rows are not fetched.

Cursor reference

To specify the cursor you wish to close, you must specify a currently open cursor using a cursor reference (see "Cursor Reference" on page 140).

SQL Delete

The **sql delete** statement removes rows from a table.

Syntax

sql delete from *table_name*
  [**where** {*search_condition* | **current of** *cursor_reference*}]
  [**on session** {*session_object_reference* | **default**}]**;**

or

{*numeric_attribute* | *numeric_variable*} = (*sql_delete_statement*);

Example

```
sql delete from artist_table where born < 1500;
```

Description

The **sql delete** statement removes the specified rows from a database table. The **where** clause identifies the particular rows to be deleted. If you do not include the **where** clause, all rows in the table are deleted.

Return Value

Number of rows deleted

If the **sql delete** statement succeeds, it returns a numeric value indicating the number of rows that were deleted. A value of 0 indicates that no rows were deleted because a matching row was not found. A value of 1 or more indicates the number of rows that were deleted.

To assign the return value to a numeric variable or attribute, you must enclose the entire **sql delete** statement in parentheses. See the *TOOL Reference Manual* for information about using SQL statements in numeric expressions.

If the **sql delete** statement fails, Forte raises an exception. The return value does not indicate whether there was an error.

Table Name

The table name identifies the database table that contains the rows to be deleted. The table must be available to the database session used for the **sql delete** statement. See “On Session Clause” below.

Where Clause

The optional **where** clause specifies the rows to be deleted. If you omit the **where** clause, all rows in the table are deleted. If you include a **where** clause, you use either a search condition, or, if your particular database system permits it, a cursor.

Using a search condition

The search condition for the **where** clause specifies a condition that deleted rows must meet. If the condition is TRUE for a row, the row is deleted. If the condition is FALSE for a row, the row is retained.

The search condition is the same as the search condition in the **where** clause of the **sql select** statement (see "SQL Select" on page 141). If you use a Forte variable or attribute in the **where** clause, you must preface it with a colon to distinguish it from a database name. The following example illustrates:

```
name : TextData = new(value = 'Leonardo da Vinci');
sql delete from artist_table where name = :name;
```

Using a cursor

For databases that allow positioned update, you can use a cursor with the **sql delete** statement to delete the single row to which the cursor is pointing (that is, the last row fetched). To identify the cursor, you must use a cursor reference. The cursor must be defined for the table specified in the **delete** statement and the cursor must include the for-updateclause.

Before you can use a cursor with the **sql delete** statement, you must use the **sql open cursor** statement in an explicit transaction. Then you use the **sql fetch** statement to position the cursor on the row you want to delete. Finally, you use the **where current of** clause in the **sql delete** statement with a reference to that cursor. Note that when you use the **where current of** clause, you cannot use the **on session** clause, because Forte uses the session associated with the cursor when the cursor was opened.

On Session Clause

The **on session** clause identifies a database session to use for the execution of this statement. You can use a service object name or a reference to an object of the DBSession class.

The **default** keyword specifies the DBSession object associated with a variable or attribute named DefaultDBSession. If you omit the **on session** clause, it is the same as specifying **default**.

SQL Execute Immediate

The **sql execute immediate** statement executes one SQL statement.

Syntax

sql execute immediate
{*string_literal* | *string_variable* | *attribute* | *TextData_reference*}
[**on session** {*session_reference* | **default**}]**;**

or

{*numeric_attribute* | *numeric_variable*} =
(*sql_execute_immediate_statement*);

Example

```
sql execute immediate 'create table xyz (co1 int)'
  on session dbsess;
```

Description

The **sql execute immediate** statement executes a single SQL statement that you enter directly as a literal string or that is stored in a Forte variable or attribute.

The **sql execute immediate** statement allows you to execute SQL statements that you cannot include directly in a method or that are constructed at runtime. For example, you can use **sql execute immediate** to execute a create table or create index statement.

Literal string

The literal string can be any SQL statement that does not return results. Be sure to enclose the string in single quotes.

String or TextData

You can use either a variable or attribute of type string or TextData to specify the SQL statement. If you use a string, its value must equal a single SQL statement. If you use a TextData object, its Value attribute must be set to a single SQL statement.

execute immediate and transactions

The **sql execute immediate** statement can be used to issue data definition language (DDL) statements, such as create table. Database systems differ in how they treat DDL statements within a transaction. They may:

- permit DDL statements within multi-statement transactions
- prohibit DDL statements within multi-statement transactions
- cause an implicit commit of the transaction in progress.

The following table summarizes the behavior of each Forte-supported DBMS:

Database	Interaction between a DDL statement and a transaction
DB2	Allows one or more DDL statements in a multi-statement transaction.
Informix	Allows one or more DDL statements in a multi-statement transaction.
Ingres	Allows one or more DDL statements in a multi-statement transaction.
ODBC	Uses the semantics of the underlying database.
Oracle	Use of a DDL statement automatically commits the current transaction.
Rdb	Allows one or more DDL statements in a multi-statement transaction.
Sybase	Allows only one DDL statement in a transaction.

If your application must be portable across databases, do not put **execute immediate** statements containing DDL in transactions that contain more than one SQL statement. Instead, enclose each DDL statement in its own explicit transaction block.

Return Value

Number of rows affected

If the **sql execute immediate** statement succeeds, it returns a numeric value indicating the number of rows that were affected. A value of 0 indicates that no rows were affected. A value of 1 or more indicates the number of rows that were affected.

To assign the return value to a numeric variable or attribute, you must enclose the entire **sql execute immediate** statement in parentheses. See the *TOOL Reference Manual* for information about using SQL statements in numeric expressions.

If the **sql execute immediate** statement fails, Forte raises an exception. The return value does not indicate whether there was an error.

On Session Clause

The **on session** clause identifies a database session to use for the execution of this statement. You can use a service object name or a reference to an object of the DBSession class.

The **default** keyword specifies the DBSession object associated with a variable or attribute named DefaultDBSession. If you omit the **on session** clause, it is the same as specifying **default**.

SQL Execute Procedure

The **sql execute procedure** statement executes a database procedure.

Syntax

sql execute procedure *procedure_name*
[([**input** | **output** | **input output**] *parameter* = *expression*
[**,** [**input** | **output** | **input output**] *parameter* = *expression*]...)]
[**on session** {*session_object_reference* | **default**}]**;**

or

DataValue_reference = (*sql _execute_procedure_statement)*;

Example

```
empid : integer = 12345;
salaryIncrement : integer = 15000;
-- Passing parameters by position.
sql execute procedure updateSalary(empid, salaryIncrement);

-- Passing parameters by name.
sql execute procedure updateSalary(AddToSalary = salaryIncrement,
   Id = empid);
```

Description

The **sql execute procedure** statement executes a database procedure using the parameter values you specify. If your database supports it, the procedure can use output parameters to return values to your Forte application.

Note

You must use this TOOL SQL statement to execute database procedures for all databases *except Sybase*; you cannot execute procedures using classes in the GenericDBMS Library. However, to process results from a Sybase stored procedure, you must use methods in the GenericDBMS classes; you cannot use this TOOL statement. See "Executing Prepared Queries" on page 87 for more information.

Return Value

Procedure return value

If the procedure produces a return value, you can assign the return value to a Forte variable or attribute. The **sql execute procedure** statement returns a DataValue object reference, which is the return value for the procedure. The actual class of the DataValue object corresponds to the return type of the procedure.

The variable or attribute to which you assign the return value must be a DataValue class, and the type must be compatible with the data type of the return value (see Chapter 4, "Working with Data Types" for information about data type compatibility). When you assign the return value to an attribute or variable, you must enclose the **sql execute procedure** statement in parentheses. See the *TOOL Reference Manual* for information about using SQL statements in expressions.

Note that the return value for this statement does not indicate whether there was an error. If the statement fails, Forte raises an exception.

Procedure Name

The procedure name identifies the database procedure to be executed. This can be any procedure available to the database session associated with the **execute procedure** statement (see "On Session Clause" below).

Parameter List

The optional parameter list provides input values for the procedure. If your procedure is defined with output or input-output parameters, you can also use this parameter list to specify variables or attributes to store the output values. Although the input, input-output, and output parameters for database procedures appear to be the same as the parameters for Forte methods, the following differences apply:

- the database contains and controls their definitions
- for database parameters, you must specify whether they are input or output at the point of call because Forte does not have the parameter definition
- you must make sure that the call matches the original parameter definition because the Forte compiler cannot check this

Do not use colons with Forte names

Note that you do not use colons to identify the Forte names in the parameter list in the **sql execute procedure** statement.

You must specify values for all required parameters (that is, parameters that do not have default values). You do not need to specify values for optional parameters (parameters having default values). Any parameter that you omit uses its default value.

Input parameters

To specify an input parameter value with an optional name, enter the name of the parameter as specified in the procedure declaration followed by the value. The value can be any expression that is compatible with the data type of the parameter (see Chapter 4, "Working with Data Types" for information about data type compatibility). The **input** key word before the parameter name is not required, because this is the default.

Output and input-output

To specify an output or input-output parameter value, use the **output** or **input output** options to identify the kind of parameter. Then enter the name of the parameter as specified in the procedure declaration followed by the Forte variable or attribute to specify the input value (for input-output parameters) and contain the output value (for both input-output and output parameters). The data type of the variable can be any simple type or DataValue type that is compatible with the data type of the parameter.

Because Forte cannot verify the parameter names or types at compile time, you get a runtime error if they are incorrect. You will also get a runtime error if you try to use the **input output** or **output** options and the parameter was not originally declared for that purpose.

On Session Clause

The **on session** clause identifies a database session to use for the execution of this statement. You can use a service object name or a reference to an object of the DBSession class.

The **default** keyword specifies the DBSession object associated with a variable or attribute named DefaultDBSession. If you omit the **on session** clause, it is the same as specifying **default.**

Exceptions

Any subclass of DBResourceException.

SQL Fetch Cursor

The **sql fetch cursor** statement allows you to fetch the next row, the next *n* rows, or the entire result set.

Syntax

sql fetch [[**next** {*integer_constant* | :*integer_varName*}] **from**]
cursor *cursor_reference*
[**into** {**:***object_reference* | *simple_list* | *array_reference*}]**;**

or

{*numeric_variable* | *numeric_attribute*} = (*sql_fetch_cursor_statement*)**;**

Example

```
sql fetch cursor dbcursor into :a;
```

Description

If you fetch one row, **sql fetch cursor** moves the cursor to the next row in the result set and assigns the values in the row to the specified attributes or variables.

If you fetch multiple rows, the **sql fetch cursor** statement fetches either the entire result set or the specified number of rows into the Forte array specified in the **into** clause. Fetching multiple rows is more efficient than fetching rows one at a time. In addition, if array support exists in the DBMS, Forte takes advantage of this.

Opening the cursor

Before you can use **sql fetch**, you must first open the cursor using the **sql open cursor** statement. After you open the cursor, the cursor is positioned before the first row in the result set.

Fetching single rows in a result set

If you are fetching rows one at a time, you can use the **sql fetch** statement in a loop to process each row in the result set. The first time you use the **sql fetch** statement, Forte moves the cursor to the first row in the result set. Then with each successive **sql fetch** statement, Forte moves the cursor forward one row, until it fetches the last row in the result set. An example follows:

Example: using **sql fetch** in a loop

```
-- Assume that the following cursor is declared.
cursor artist_cursor (name : string)
begin
  select name, country, born, died, school, comments
  from artist_table
  where name LIKE :name;
end;
-------------------------------------------------------
-- The following code will use the cursor.
dbsess : DBSession = ... more here ...
dbcursor : artist_cursor;
a : Artist = new;
painters : Array of Artist = new;
sql open cursor dbcursor ('%') on session dbsess;
while (sql fetch cursor dbcursor into :a)
     > 0 do
  -- Since 'a' will be reused, make a new one after append.
```

```
  painters.AppendRow(a);
  a = new;
end while;
sql close cursor dbcursor on session dbsess;
-- Now print them out
for x in painters do
  x.WriteToLog();
end for;
```

Specifying a number of rows to fetch

If you use the optional key word **next**, you can specify that a given number of rows should be fetched, and you must use an array reference in the **into** clause.

If the value following the **next** key word (say *n*) evaluates to a positive number, at most *n* rows are fetched from the cursor. If less than *n* rows remain, they are all fetched. If *n* evaluates to 0, all remaining rows are fetched from the cursor. If *n* evaluates to less than 0, no rows are fetched. You can use the Array.Items attribute to find out how many rows were fetched and put into the array.

An example showing the use of the **next** keyword follows:

```
dbsess  :  DBSession;
rows    : integer = 1;
dbcursor :  artist_cursor;
painters :  array of artist  = new;

sql open cursor dbcursor ('%') ON SESSION dbsess;
while (rows > 0) do
    sql fetch next 10 from cursor dbcursor into :painters;
    rows = painters.Items;
    -- do something with painters ...
end while;
```

Transactions

The **sql fetch cursor** statement is never enclosed in an implicit transaction. The behavior of a **sql fetch cursor** statement executed outside of an explicit TOOL transaction depends on when the cursor was opened. If the cursor was opened in an explicit TOOL transaction, then it was closed at the end of the transaction and any fetch outside the transaction in which it was opened will fail. If the cursor was opened outside of an explicit transaction, the result set for the cursor was buffered and the **fetch** statement will fetch the next row from the buffer.

Return Value

Number of rows fetched

The **sql fetch cursor** statement returns a numeric value indicating the number of rows that were fetched. A return value of one indicates that Forte successfully fetched one row. A return value of greater than one indicates that you fetched the entire result set into an array, and the number indicates the total number of rows in the array. A return value of zero indicates that there are no more rows left in the result set.

To assign this return value to a numeric variable or attribute, you must enclose the entire **sql fetch cursor** statement in parentheses. See the *TOOL Reference Manual* for information about using SQL statements in numeric expressions.

Note that the return value for this statement does not indicate whether there was an error. If the statement fails, Forte raises an exception.

Cursor

To identify the cursor, you must use the cursor reference. This must be a cursor that you have opened with the **sql open cursor** statement.

Into Clause

The **into** clause identifies the attributes and/or variables in which you want to store the column values. You can specify either a single object reference, or a list of simple data items or DataValue objects. Note that when you use Forte names in a SQL statement, you must precede each name with a colon.

Object reference

You can reference an existing object in which to store the column values. The attributes in the object must correspond by name with the columns in the result set. They must also have compatible data types. If there are any attributes in the object that have class types (other than DataValue types), Forte ignores these, even if they have matching names.

If there are more columns in the row than attributes in the object, the extra columns are ignored (you cannot access them from TOOL). If there are more attributes in the object than columns in the row, the extra attributes retain the values they had before the **sql fetch** statement.

The object must already have been created, and any attributes that are DataValue objects must be created before the **sql fetch** statement.

Note that if you wish to fetch the values from a table into an object whose attribute names do not match the column names, you can use the column list of the **select** statement for the cursor to specify correlation names for the columns.

Array object reference

If the object you reference is an array object, the **sql fetch** statement fetches either the entire result set in the cursor or the number of rows specified in the **next** clause into the array. If the array reference is NIL, Forte creates a new array object and adds the rows in the result set to the array. If the array reference is for an existing array object, Forte replaces the values of the attributes in each of the array rows with the values from the result set. If there are more rows in the result set than in the original array, Forte adds the remaining rows as new rows. If there are more rows in the array, Forte deletes the extra array rows.

Simple data items

The list of simple attributes and/or variables can include any attributes or variables with a simple data type or with a subclass of the DataValue class (such as TextData, IntegerNullable, or DateTimeData). The order of the attributes and/or variables must match the order of the columns in the result set, and they must have compatible data types (see Chapter 4, “Working with Data Types” for information on data type compatibility). The syntax is:

variable | *attribute* [**,** *variable* | *attribute*]...

If you include DataValue objects on the list of simple data items, you must create them before using the **sql fetch** statement.

In order to return null values from the database, you must assign the value to a nullable DataValue object. If you do not use a nullable DataValue object, you will get a runtime exception. Use the IsNull method to check if there is a null value. See the Forte online Help for information on the nullable DataValue classes.

SQL Insert

The **sql insert** statement adds a new row to a database table.

Syntax

sql insert into *table_name* [(*column* [**,** *column*]...)]
{**values** ({*object_reference* | *simple_list*}) | *select_statement*}
[**on session** {*session_object_reference* | **default** }]**;**

or

{*numeric_attribute* | *numeric_variable*} = (*sql_insert_statement*);

Example

```
a : Artist = new ... fill in data...;
sql insert into artist_table
  (name, country)
  values (:a.name, :a.country)
  on session dbsess;
```

Description

The **sql insert** statement adds a new row to the specified table. You can either provide a list of values to use for the new row or you can use a **select** statement to get the values from another table. If you wish to add a set of rows stored in a Forte array, you can specify an array object in the **values** clause.

Return Value

Number of rows inserted

The **sql insert** statement returns a numeric value indicating the number of rows that were inserted. A value of zero indicates that no rows were inserted. A value of one indicates that one row was inserted. If you are inserting an set of rows stored in a Forte array, the return value indicates the total number of rows added to the table.

To assign this return value to any numeric variable or attribute, you must enclose the entire **sql insert** statement in parentheses. See the *TOOL Reference Manual* for information about using SQL statements in numeric expressions.

Note that the return value for this statement does not indicate whether there was an error. If the statement fails, Forte raises an exception.

Table Name

The table name identifies the database table to which you wish to add a new row. This can be any table available from the database session that you are using with the **sql insert** statement.

The Column List

The column list identifies the columns for which you are providing values. Any columns that you do not include on this list will be set to their default values. If you are specifying the insert values with a list of simple variables and attributes, or with a **select** statement, the column list must match *by position* with the insert values. If you are specifying the insert values with an object reference, the column list must match the attributes in the object *by name*.

Specifying the Insert Values

You can specify the values for the new row either by using the **values** clause to enter a list of values or by using an **sql select** statement to select the values from another table.

values clause

The **values** clause provides the values to use for the new row. You can use a single object reference or a list of variables and attributes.

Object reference

You can reference an existing object to provide the column values. The attributes in the object must correspond by name with the columns in the table (or the columns you specified in the column list). They must also have compatible data types. Any attributes that do not have simple data types or DataValue class types are ignored.

If you provided a column list for the statement, any attribute names that are not also on the column list will be ignored.

Array object reference

By specifying an array object in the **values** clause, you can insert an entire set of rows with a single **sql insert** statement. Forte also uses any underlying optimizations that may be provided by a specific vendor. To match the attributes in the array rows with the columns in the table, Forte uses name matching as described above for a single object.

Simple data items

The list of simple attributes and/or variables can include any attributes or variables with a simple data type or with a subclass of the DataValue class (such as the TextData, IntegerNullable, or DateTimeData classes). The order of the attributes and/or variables must be in the same order as the columns in the column list of the **sql insert** statement, and they must have compatible data types (see Chapter 4, "Working with Data Types" for information on data type compatibility). The syntax is:

variable | *attribute* [**,** *variable* | *attribute*]...

Using the **select** statement

The **select** statement selects a row from another table and uses its values for the new row. You can use any valid **select** statement, without an **into** clause. The columns in the **select** statement's column list must be in the same order as the columns in the table you are inserting into. If you specify an asterisk instead of a column list, the columns in the table you are selecting from must be in the same order as the table you are inserting into.

If the **select** statement selects more than one row, the **sql insert** statement uses the values from the first row.

On Session Clause

The **on session** clause identifies a database session to use for the execution of this statement. You can use a service object name or a reference to an object of the DBSession class.

The **default** keyword specifies the DBSession object associated with a variable or attribute named DefaultDBSession. If you omit the **on session** clause, it is the same as specifying **default**.

SQL Open Cursor

The **sql open cursor** statement selects rows from a database table to be used with a database cursor.

Syntax

sql open cursor *cursor_reference*
[(*expression* [, *expression*]...)]
[**on session** {*session_object_reference* | **default**}]**;**

Example

```
dbcursor : empcursor;
empid : integer;
sql open cursor dbcursor (empid) on session dbsess;
```

Description

The **sql open cursor** statement executes the **select** statement associated with the cursor and positions the cursor before the first row in the result set.

For an overview of defining and using cursors, see "Selecting Multiple Rows using Cursors" on page 75.

Positioned update

When you define a cursor in the Cursor Workshop, you may define it as a *read-only* or *for-update* cursor. The effect of using either type of cursor varies by database vendor. One use of a for-update cursor is to use it in a "positioned update." A positioned update uses the **where current of** clause in the **sql delete** or **sql update** statements to update (or delete) the current row. Of the database vendors, Oracle, Ingres, Informix, and Rdb support positioned update of cursors; Sybase, DB2 and ODBC do not support positioned updates.

As an alternative to using cursors, you can use the TOOL **for** statement to work with a result set of a **select** statement. The **for** statement is a simpler way to work with ad hoc queries; a cursor is more efficient for queries that will be re-executed, and the cursor can be shared. See the *TOOL Reference Manual* for information on the **for** statement.

Use an explicit transaction

For best performance, you should start an explicit Forte transaction, using the begin transaction TOOL statement, before opening a cursor. While you can use **sql open cursor** outside of an explicit transaction, it is not recommended, as explained below.

Effect of implicit transactions

An **sql open cursor** statement executed outside an explicit Forte transaction is enclosed in an implicit transaction. Since the implicit transaction ends before the data is actually fetched, and because most DBMSs do not allow cursors to remain open when a transaction ends, the session automatically buffers the data as part of executing the **sql open cursor** statement. This means that the entire result set is retrieved for the cursor. Subsequent **sql fetch cursor** statements retrieve results from the cursor as usual, but the data comes from the buffer rather than from the database.

Explicit transactions are preferred for updating

An implicit transaction for an open cursor buffers all the data and then automatically commits the transaction in the database. The actual data in the database corresponding to the buffered data can be concurrently updated without any notification to your program nor any locking on the data. Concurrent updating does not pose a problem for read-only data; however, to assure maximum consistency for data that will be updated, you should use explicit transaction statements.

Cursor Reference

When you work with cursors, you use a cursor reference. After a cursor is defined in the Cursor Workshop it has a cursor name. However, to reference a cursor in a TOOL SQL statement, you declare a cursor reference variable using the following format:

variable_name : *cursor_name*

Subsequently, in SQL statements that use that same cursor, you use the cursor reference.

Placeholder Assignment

The placeholder assignment list specifies the values for the placeholders used in the original cursor declaration. To specify these values, simply enter a list of values. The order of the values must match the order specified in the original cursor definition by position. The value for a placeholder can be any expression that is compatible with the data type of the placeholder.

Do not use colons with Forte names

Note that you do not use colons to identify the Forte names in the placeholder list in the **sql open cursor** statement. In the following example, colons do not precede the Forte variables empid, ival, and fval.

Example: placeholder assignment

```
sql open cursor dbcursor (empid) on session dbsess;
integer ival = 4;
float fval = 5.2;
sql open cursor mycursor (ival, fval) on session dbsess;
```

On Session Clause

The **on session** clause identifies a database session to use for the execution of this statement. You can use a service object name or a reference to an object of the DBSession class.

The **default** keyword specifies the DBSession object associated with a variable or attribute named DefaultDBSession. If you omit the **on session** clause, it is the same as specifying **default**.

SQL Select

The **sql select** statement retrieves one or more rows from one or more database tables.

Syntax

sql select [**all** | **distinct**] {* | *column_list*}
[**into** {*object_reference* | *simple_list*}]
[**from** *table_name* [, *table_name*]...]
[**where** *search_condition*]
[**group by** *column_name* [, *column_name*]...]
[**having** *search_expression*]
[**order by** *column* [**asc** | **desc**] [, *column* [**asc** | **desc**]]...]
[**on session** {*session_object_reference* | **default**}]**;**

or

{*numeric_attribute* | *numeric_variable*} = (*sql_select_statement*);

Example

```
sql select * into :artist_object from artist_table
where name = :vname
on session dbsess;
i : integer;
vname, vcountry : TextData = new;
i = (sql select name, country into :vname, :vcountry from
      artist_table on session dbsess);
```

Description

The **sql select** statement allows you to retrieve rows from one or more database tables, and to store the values in Forte variables or attributes.

You can use this statement to select a single row into a Forte object or set of simple variables, or to select a set of rows to insert into a Forte array. You can also use the **sql select** statement in a **for** statement and in a cursor declaration. (See the *TOOL Reference Manual* for information on using the **sql select** statement in a **for** statement. See "Repeating a Statement Block" on page 77 for information about using **sql select** in the **cursor** statement.)

Unless you are selecting into an array object (see "Into Clause" on page 143), if you use **sql select** on its own, you can select only one row. If the result is more than one row, no data is returned and Forte raises a DBOperationException.

Note that this section does not contain an exhaustive list of the valid syntax for every clause of the **select** statement for all databases. Generally speaking, Forte supports almost all standard and vendor-specific supporting clauses, such as group by, compute, and so on.

Return Value

Number of rows selected

The **sql select** statement returns a numeric value indicating the number of rows that were selected. A value of zero indicates that there was no matching row. A value of one or greater indicates the total number of matching rows.

To assign this return value to any numeric variable or attribute, you must enclose the entire **sql select** statement in parentheses. See the *TOOL Reference Manual* for information about using SQL statements in numeric expressions.

Note that the return value for this statement does not indicate whether there was an error. If the statement fails, Forte raises an exception.

Eliminating Duplicate Rows

distinct and **all** options

When you are using a **for** loop or a cursor to select a set of rows, or if you are selecting into an array, you can use the **distinct** option to eliminate duplicate rows from the result set. This option ensures that the statement does not return identical rows. The **all** option specifies the default, which is that the statement returns all rows that match the **where** clause.

Column List

You must specify which columns you wish to retrieve. You can use the asterisk (*) to specify all columns or you can provide a list of column names. We recommend that you do specify the column names, even if you want to retrieve them all. Forte can only optimize the **select** statement when the column names are specified at compile time.

The syntax for the column list is:

column [*alias*] [**,** *column* [*alias*]]...

If you specify a list of simple attributes and/or variables in the **into** clause (see below), these must match the columns in the column list by position. If you specify an object in the **into** clause, the attributes in the object must match the columns in the column list by name. However, if the object you wish to store the values in has attributes with different names than the columns, you can use the optional alias in the column list to "rename" the column.

In the following example, the Employee class has the following attributes: empId, EmpFname, and empLname. The columns in the database table have the following names: id, firstName, and lastName. By using aliases in the column list, the **sql select** statement provides the matching names to associate each column with the appropriate attribute in the object.

Example: renaming columns

```
sql select id empId, firstName empFname,
    lastName empLname from emptable
  into :empObject
  where id = :id;
```

Note that the use of column renaming is subject to the restrictions of your database. For example, in Rdb you rename columns using the ANSI SQL "AS" clause.

Using aggregates and functions

You can include aggregates and functions in the column list. Forte lets you use any SQL construct that is supported by your particular DBMS. However, because aggregates and functions vary between databases, this may mean that your SQL is not portable.

Into Clause

The **into** clause identifies the attributes and/or variables in which you want to store the column values. You can specify a single object or a list of simple data items.

Object reference

You can reference an existing object in which to store the column values. The attributes in the object must correspond by name with the columns in the column list (or the columns in the table if you use an asterisk). They must also have compatible data types (see Chapter 4, "Working with Data Types" for information on data type compatibility). If there are any attributes in the object whose type is a class (other than DataValue types), Forte ignores these, even if they have matching names.

If there are more columns in the column list than attributes in the object, the extra columns are ignored (you cannot access them from TOOL). If there are more attributes in the object than columns in the column list, the extra attributes retain the values they had before the **sql select** statement.

The object must already have been created, and any attributes that are DataValue objects must be created before the **sql select** statement.

Array object reference

If the object you reference is an array object, the **sql select** statement fetches the entire result set into the array. If the array reference is NIL, Forte creates a new array object and adds the rows in the result set to the array. If the array reference is for an existing array object, Forte replaces the values of the attributes in each of the array rows with the values from the result set. If there are more rows in the result set than in the original array, Forte adds the remaining rows as new rows. If there are more rows in the array than in the result set, Forte deletes the extra rows.

Simple data items

The list of simple attributes and/or variables can include any attributes or variables with a simple data type or with a subclass of the DataValue class (such as TextData, IntegerNullable, or DateTimeData). The order of the attributes and/or variables must match the order of the columns in the result set, and they must have compatible data types (see Chapter 4, "Working with Data Types" for information on data type compatibility). The syntax is:

variable | *attribute* [**,** *variable* | *attribute*]...

If you include DataValue objects on the list of simple data items, you must create them before using the **select** statement.

From Clause

The **from** clause identifies one or more database tables from which you wish to select rows. You can include any tables available to the database session that you are using with the **sql select** statement.

Where Clause

The **where** clause specifies one or more conditions that the selected row or rows must meet. This can be any search condition that is allowed by your database management system.

Using Forte attributes and variables

You can use Forte attributes and variables in the search condition to specify values. These are allowed anywhere that a literal is allowed. Be sure to precede all Forte names with colons to distinguish them from column names.

Using functions

You can include any SQL functions in your search condition that are allowed by your particular database. However, since supported functions vary among database systems, this may mean that your SQL is not portable.

If you omit the **where** clause, the **select** statement returns all the rows in the table.

Group By Clause

When you are using aggregates in the column list, you can use the **group by** clause to break the result set into groups. Forte produces a summary value for each group rather than for the result set as a whole.

The **group by** clause specifies the columns to use as the basis for creating the groups. You can enter any columns from the **select** statement's column list. Forte starts a new group each time the value changes in the specified column.

Having Clause

The **having** clause is for use with the **group by** clause. It specifies a condition that rows must meet in order to be included in the groups controlled by the **group by** clause. The search condition for the **having** clause is exactly the same as the search condition for the **where** clause, except that it can also include aggregates.

Order By Clause

The **order by** clause sorts the result set by one or more columns. You can specify any columns from the **select** statement's column list.

For each sort column you can specify an ascending sort or descending sort. If you do not specify either, the default is ascending.

On Session Clause

The **on session** clause identifies a database session to use for the execution of this statement. You can use a service object name or a reference to an object of the DBSession class.

The **default** keyword specifies the DBSession object associated with a variable or attribute named DefaultDBSession. If you omit the **on session** clause, it is the same as specifying **default**.

Exceptions

Any subclass of DBResourceException.

SQL Update

The **sql update** statement changes values in one or more rows from a database table.

Syntax

sql update *table_name* **set** *column* = *expr* [**,** *column* = *expr*]...
  [**where** {*search_expression* | **current of** *cursor_reference*}]
  [**on session** {*session_object_reference* | **default**}]**;**

or

{*numeric_attribute* | *numeric_variable*} = (*sql_update_statement*);

Example

```
sql update artist_table set born = :vborn
  where name = :vname on session dbsess;
```

Description

The **sql update** statement replaces the current column values in the selected row or rows with the new values that you specify. You use the **set** clause to specify the new values for the columns. You use the **where** clause to select the rows that you wish to update. Without the **where** clause, the **sql update** statement changes the specified column values in all the rows in the table.

Return Value

Number of rows updated

The **sql update** statement returns a numeric value indicating the number of rows that were updated. A value of zero indicates that no rows were updated. A value of one or more specifies the total number of rows that were updated.

To assign this return value to any numeric variable or attribute, you must enclose the entire **sql update** statement in parentheses. See the *TOOL Reference Manual* for information about using SQL statements in numeric expressions.

Note that the return value for this statement does not indicate whether there was an error. If the statement fails, Forte raises an exception.

Table Name

The table name identifies the database table that you wish to update. This can be any table from the database session that you are using with the **sql update** statement (see "On Session Clause" on page 146).

Set Clause

To specify the new values for the rows, you enter a list of column names with their corresponding values.

Column values list

The value for a column can be any expression that is compatible with the data type of the column (see Chapter 4, "Working with Data Types" for information on data type compatibility). If you use Forte names in the expressions, be sure to precede the names with colons.

Where Clause

The **where** clause selects the rows to be updated. You can specify a search condition or, if your particular database system permits it, a cursor. If you omit the **where** clause, all the rows in the table are updated.

Specifying a search condition

The search condition for the **where** clause specifies a condition that the rows must meet. Forte tests all the rows in the table. If the condition is TRUE for a row, the row is updated. If the condition is FALSE for a row, the row is not updated.

The search condition is the same as the search condition in the **where** clause of the **sql select** statement (see "Where Clause" on page 143). If you use Forte variable or attribute names in the **where** clause, be sure to preface them with colons. This distinguishes the Forte names from the column names.

Example: using the **where** clause

```
name : string = 'Leonardo da Vinci';
whenborn : integer = 1922;
sql update artist_table set born = :whenborn
  where name = :name on session dbsess;
```

Specifying a cursor

For databases that allow positioned update, you can use a cursor with the **sql update** statement to update the single row to which the cursor is pointing (that is, the last row fetched). To identify the cursor, you must use a cursor reference. The cursor must be defined for the table specified in the **update** statement, it must include the for-update clause, and if updating is restricted to certain columns, you can update only those columns.

Before you can use a cursor with the **sql delete** statement, you must use the **sql open cursor** statement in an explicit transaction. Then you use the **sql fetch** statement to position the cursor on the row you want to update. Finally, you use the **where current of** clause in the **sql delete** statement with a reference to that cursor. Note that when you use the **where current of** clause, you cannot use the **on session** clause, because Forte uses the session associated with the cursor when the cursor was opened.

On Session Clause

The **on session** clause identifies a database session to use for the execution of this statement. You can use a service object name or a reference to an object of the DBSession class.

The **default** keyword specifies the DBSession object associated with a variable or attribute named DefaultDBSession. If you omit the **on session** clause, it is the same as specifying **default.**

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W